FreeBSD™ JOURNAL

July/August 2017

Lynx
Network Traffic Analysis

Getting Started with
iohyve

OpenRC
An Interview with Joe Malony

Understanding
DOCKER
For FreeBSD Users

BORN TO DISRUPT

MODERN. UNIFIED. ENTERPRISE-READY.

INTRODUCING THE TRUENAS® X10, THE MOST COST-EFFECTIVE ENTERPRISE STORAGE ARRAY ON THE MARKET.

Perfectly suited for core-edge configurations and enterprise workloads such as backups, replication, and file sharing.

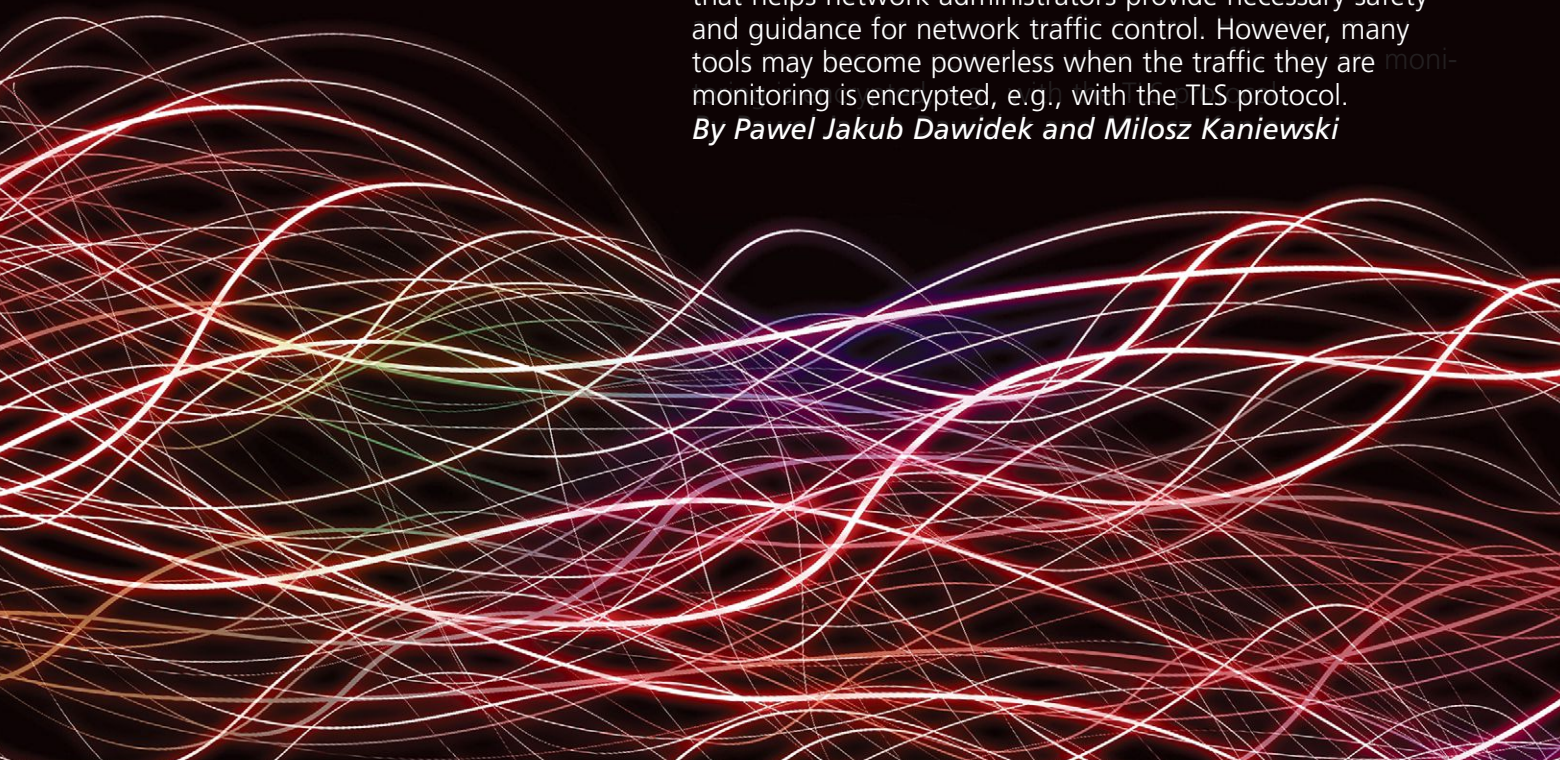
- ★ **Modern:** Not based on 5-10 year old technology (yes that means you legacy storage vendors)
- ★ **Unified:** Simultaneous SAN/NAS protocols that support multiple block and file workloads
- ★ **Dense:** Up to 120 TB in 2U and 360 TB in 6U
- ★ **Safe:** High Availability option ensures business continuity and avoids downtime
- ★ **Reliable:** Uses OpenZFS to keep data safe
- ★ **Trusted:** Based on FreeNAS, the world's #1 Open Source SDS
- ★ **Enterprise:** 20TB of enterprise-class storage including unlimited instant snapshots and advanced storage optimization for under $10,000

The new TrueNAS X10 marks the birth of a new entry class of enterprise storage. Get the full details at iXsystems.com/TrueNAS.

iXsystems™

Table of Contents

July/August 2017



Support FreeBSD®

Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.
freebsdfoundation.org/donate

FreeBSD™
FOUNDATION

S&W Publishing LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Copy Editor • Annaliese Jakimides

Art Director • Dianne M. Kischitz
dianne@freebsdjournal.com

Office Administrator • Michael Davis
davism@freebsdjournal.com

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
5757 Central Ave., Suite 201, Boulder, CO 80301
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org

LETTER from the Board

Welcome to the late-summer issue of *FreeBSD Journal*, at least, late summer from where I'm sitting, in Cambridge, England. Next week will be BSDCam, a FreeBSD Developer Summit that now occurs every August, when the students are gone and the professors run wild, or as wild as professors are bound to run. I'm sure we'll have more to say about BSDCam after the summit is over, but now, let's look at what we've got for you to read while you relax at the beach, or wherever you spend your time reading the *Journal*.

It's probably nearly impossible to read about virtualization and operating systems in the last year or two and not come across references to Docker. As with any new technology the next logical question is, "What about FreeBSD?" Kurt Lidl answers that question with an overview of Docker and FreeBSD, not only covering the current state, but also looking to the future of FreeBSD and Docker. Next, Benedict Reuschling, *FreeBSD Journal* and FreeBSD Foundation Board Member, interviews Joe Maloney from the TrueOS project about the new OpenRC system being deployed with the latest releases of TrueOS. Pawel Jakub Dawidek and Milosz Kaniewski tell us about how they're using FreeBSD to do Network Traffic Analysis to look into TLS-encrypted flows. Many people are familiar with bhyve, the native hypervisor on FreeBSD, as well as ZFS, the zetabyte filesystem. Trent Thompson writes about iohyve, which combines these two technologies to make your systems deployment life a little easier.

We round out the issue with three columns: New Faces, where Dru Lavigne chats with two of the newest ports committers; svn Update, where Steven Kreuezer shows us all that's new and shiny in FreeBSD; and a conference report from BSDCan, by Roller Angel. If you want to know what's coming up next for FreeBSD events, look no further than Dru Lavigne's Events Calendar which will tell you when and where to find FreeBSD folk meeting worldwide.

George Neville-Neil
President of the *FreeBSD Foundation*
Board of Directors

SEE TEXT ONLY

GETTING STARTED WITH bhyve

By Trent Thompson

If you've ever had to navigate the system administration waters, you've probably come across virtualization as a way to make your life a little easier. Instead of having racks full of computer hardware, virtualization allows you to essentially turn one computer into many computers, thus saving precious space and computing cycles. Virtualization can help with small tasks, such as installing VirtualBox or VMWare on a workstation to test drive the new hottest Linux distribution. The tech giant Amazon uses Xen virtualization to power its Amazon Web Services cloud product, which made them over $12 billion in 2016.

Of course, the purpose of this article isn't to make you a billionaire but to help make your life a little easier. We're going to do this by leveraging two key technologies built into FreeBSD: ZFS and bhyve.

iohyve's Origin

ZFS is the filesystem and logical volume manager that has been in FreeBSD since FreeBSD 7 in 2008, when it was still considered experimental. It originated at Sun Microsystems (now Oracle) in 2005, when it was released with the OpenSolaris operating system, and since then has been ported to many other operating systems. Key features in ZFS include automatic data integrity, software RAID (called RAID-Z), creating logical volumes that act as raw disks (Zvols), and snapshots (and the ability to roll back to previous snapshots). We'll get into why these features are important to iohyve later.

Of course, we also need some sort of virtualization technology to help turn one computer into many computers. Indeed, FreeBSD has had jails since about FreeBSD 4 (longer than ZFS support). FreeBSD jails provide a lightweight way to virtualize the FreeBSD userland, meaning you can only virtualize FreeBSD, as all of your jails will share the same FreeBSD kernel. It's a great way to turn one FreeBSD host into many FreeBSD "virtual machines," each with their own IP addresses and separation from each other (one jail cannot access data in another jail by default, and by design). Solaris has a similar technology called Solaris Zones, which is still used today by virtualization solutions like SmartOS (based on OpenSolaris). Linux also has its own containers, provided by LXC and even Docker. The problem with most of these solutions is that they cannot virtualize other operating systems. For instance, you can't run Linux in a jail, or FreeBSD in a Docker container.

This is where hardware-assisted virtualization comes in to help. Using CPU technologies like VT-x on Intel CPUs or AMD-V on AMD CPUs, a special software suite called a hypervisor allows one operating system to virtualize another. Note that I didn't use the word "emulate." An emulator, such as QEMU, only emulates a processor, while a hypervisor connects a virtual machine to the CPU itself, thus having huge performance increases over emulation. There are a few different solutions that I've mentioned, including Xen. Xen is a powerful hypervisor, and is even available on FreeBSD. Since its beginnings around 2003, it has grown considerably. Linux distributions have a built-in hypervisor called KVM (Kernel-based Virtual Machine). Linux KVM has been around since 2007, and is even used by SmartOS to virtualize other operating systems where Zones won't cut it. The hypervisor that iohyve uses is bhyve, which has been built into FreeBSD since FreeBSD 10 in 2014. You'll note that this is considerably young in comparison to the other hypervisors mentioned. The bhyve hypervisor is a much more lightweight hypervisor, with its kernel module and userland utilities taking up less than 500k of space. The bhyve hypervisor has also been ported to other operating systems including Mac OS X where it is called xhyve, and is used by Docker to do the heavy lifting for their Linux containers.

The bhyve hypervisor may be lightweight, but it is a powerful tool with many features. A few years ago while navigating my own system administration waters, I needed to virtualize some Linux servers. After trying many different solutions, I landed on FreeBSD, mainly for the ZFS support. At the time, I was using Oracle VirtualBox with phpVirtualBox as a GUI to remotely administer virtual machines. I configured VirtualBox to store all the virtual machines on a single ZFS dataset. This worked okay, as I would be able to make snapshots of the dataset to make backups, but I wasn't able to store virtual machines on separate ZFS datasets, which, to me, made more sense. That way I can make snapshots of individual virtual machines and not a snapshot of all the virtual machines. For a very short time, I tried to modify VirtualBox to create a new dataset to put a virtual machine into, but this turned out to be more work than I wanted to put in. Around that same time, I was playing with FreeBSD jails, and was even using them to separate phpVirtualBox from the VirtualBox host so if someone exploited the PHP webserver, they would be stuck in the jail, and not on the VirtuaBox host itself. At first, I used ezjail-admin to implement this, but quickly dropped it for iocage, a jail manager that utilized ZFS in a way that was similar to what I wanted to do with VirtualBox (each jail is on its own ZFS dataset).

As previously mentioned, jails are wonderful, but they can only "virtualize" FreeBSD, not other operating systems like Linux. I really loved the idea of iocage though. Not only did each jail have its own dataset, but it also stored the properties of each jail in the ZFS user properties of the dataset, essentially eliminating the need for configuration files or a dependency on a database. I was also attracted to the iocage project because they used shell scripting, meaning that the code was easily understood at a glance, and didn't require any compilation. The iocage project has since moved

on from these two attractions, mainly due to its enormous growth since then, where they have opted for UCL configuration to speed things up (ZFS user properties queries can be very slow on a busy system) and have switched to Python to alleviate the headaches of writing a massive shell script. At first, my idea was to add bhyve support to iocage, but it quickly became clear that this was like putting a square peg in a round hole, while the size of the round hole was constantly being changed (with iocage growing at such a fast rate, it was hard to keep up with the changes).

Eventually one afternoon, I hacked out a basic script that mimicked the simple command line interface of iocage to mate bhyve and ZFS together. In the beginning, it consisted of just one script, and had very limited abilities like only support for FreeBSD virtual machines, and support for only one virtual disk that was stored as a file. I uploaded the script as a text file to GitHub's Gist repository and showed it to a handful of people. They all wanted more, and so I eventually created an actual GitHub repository, which is still around. This allowed me to turn a hacked-out script into a full-fledged project, with a wiki that can be used like a handbook, an issue tracker to help with bug reporting, and the ability for anyone to help contribute with pull requests. Since its creation, iohyve has grown with more and more features and bug fixes being added along the way. One defining moment in iohyve history was the suggestion to use ZFS ZVOLs as a way to better utilize ZFS. ZVOLs can be created and snapshotted just like regular datasets; however, they appear to the operating system as basically disk devices. This is very similar to the way many Linux KVM and Xen solutions store their virtual machines on LVM partitions, cutting some performance overhead. Eventually, the ability to also run the other BSDs, like NetBSD and OpenBSD, as well as various Linux distributions, had support in iohyve with the addition of grub2-bhyve support, which essentially acts like the GRUB bootloader for bhyve (as the name implies). This is because bhyve doesn't have a built-in BIOS to load the important bits of the operating system. There was bhyveload which will load the FreeBSD-based operating systems, and grub-bhyve, to help boot operating systems that can use GRUB to boot.

Eventually, bhyve gained support for UEFI booting, meaning there was no longer a need to use bhyveload or grub-bhyve, as the UEFI firmware could be used to load the operating system, just like many modern computers use to load operating systems today on "bare metal." With this UEFI firmware, you can run many different types of operating systems, including modern versions of the Microsoft Windows operating system family. Another recent feature addition to bhyve was the ability to have a graphical console provided by UEFI-GOP. Originally, bhyve only had support for an emulated serial console, meaning the virtual machines didn't have support for a GUI with a mouse and keyboard, like Oracle VirtualBox's remote display feature. Since FreeBSD 11, bhyve has come built in with this UEFI-GOP support as a basic VNC server. Since BSDCan 2016, iohyve has had the ability to utilize this VNC server to help install operating systems that have graphical installers like CentOS and Windows. Although that's a cool feature, we won't get much into UEFI-booted bhyve virtual machines.

How to Use iohyve

So enough of the boring iohyve origin story. Let's learn how to actually use iohyve! For the purposes of this tutorial, I will be using FreeBSD 11 Release. Although most of the hottest new features (and feature branches) of bhyve are done on the current branch, I chose to go with a release simply for the ability to keep updates easy with freebsd-update and pkg. Your mileage may vary. An important part of iohyve, ZFS, can be installed and set up during the installation process. Even though you may want to keep your iohyve virtual machines, or "guests," on a separate zpool, I still chose to go with FreeBSD installed on ZFS. If you have the overhead to run virtual machines, you have the overhead to run ZFS (terms and conditions may apply). I often chose to lump all the disks I have in a box (even if it's just one) into a zpool and install FreeBSD onto it (usually as zroot). Once the installation is done, you are given the opportunity to drop to a shell to make any changes before rebooting into your new and fresh FreeBSD installation. I take this opportunity to install some dependencies and utilities to help with making the new installation ready to go. Your "stack" may be different, but this is just one I find useful to get started. Here's the one-liner that uses pkg:

```
pkg install sudo nano tmux git htop bhyve-firmware grub2-bhyve
```

The use of `sudo` is pretty straightforward: bhyve requires root (for now) and I don't trust users. Instead of giving them access to the root account, I delegate this through `sudo`. I know

some of you are giving me weird looks with the use of nano, but it's what I've been using to edit config files since I've been editing config files. Sure, edit in base is simpler than vi, but nano's interface is more muscle memory than thinking at this point. Next on the list is tmux, the terminal multiplexor. This is handy for a number of reasons, the first of which is: you can keep your sessions opened even if your SSH session dies. Since bhyve doesn't utilize a graphical console by default, all communication can be done over null modems (like a serial connection). The use of tmux allows for you to open a new tmux window or pane for each new console to an iohyve guest. I'll go into how it can be used to help monitor your iohyve host. I like to use git because even if you can install iohyve via ports or pkg, you can make sure you're getting the latest and greatest bug fixes from the iohyve GitHub master branch. I also install htop honestly to get the cool CPU graph output, for better resource monitoring at a glance. The bhyve-firmware package installs the bhyve UEFI firmware. We won't be going into that, but it's good to have on hand just in case. You can find more info on iohyve and UEFI in the man page. Lastly is grub2-bhyve, which allows us to run other BSDs or Linux distributions as iohyve guests.

After everything is installed, I add my default user to the sudoers file and reboot into the new installation. From here on out, I use SSH to connect to the machine. In theory, this can all be done via the console as well. The first thing I do after logging in is start a new tmux session with simply:

```
tmux
```

I should warn you, if you've never used tmux before, it's awesome, and is kind of like GNU screen. Next, I install a new copy of iohyve from GitHub with:

```
git clone
https://github.com/pr1ntf/iohyve.git
```

You can also fetch the latest master ZIP file if you look for it. There are also releases available, that are also available as a port and package. Now we can install it (if we are using sudo) with:

```
cd iohyve/
sudo make install clean
cd ~
```

Voila! The magic of makefiles moves everything where it needs to be, including a man page entry and an RC script. Now we need to set up iohyve itself. One setup command only needs to be run once; the other needs to be run anytime you start your iohyve host. This can be easily done with the RC script, but we'll get to that shortly. First, we set up iohyve on a zpool. In the example, I use the built-in zroot pool. You can use whatever pool you've set up.

```
sudo iohyve setup pool=zroot
```

Next, we want to set up required kernel modules and networking for our iohyve host. You can do this manually yourself, but if your iohyve host is just going to be used for iohyve, I suggest using this method. Note this method doesn't work with wireless. There is some documentation out there on using iohyve with wireless, but for our purposes, we are going to set up networking attached to an Ethernet device. All of your iohyve guests will be attached to a bridge, and that bridge is attached to an interface. There are more complicated setups than this, but those features are new to iohyve, and bugs are still being worked out as of this writing. Check the man page for more info. In the following example, I am going to use the em0 interface on my iohyve host. You can see on which interfaces you have working networking by a simple ifconfig.

```
sudo iohyve setup kmod=1 net=em0
```

You can make sure iohyve does this on every boot by adding the following to your rc.conf:

```
iohyve_enable="YES"
iohyve_flags="kmod=1 net=em0"
```

Next, we are going to set up our first iohyve guest. We'll create the guest, then change some properties of the guest so that we can run Ubuntu Linux. Next, we'll fetch an Ubuntu ISO directly to iohyve's local ISO repository. Any installation ISO needs to be in the iohyve repository, and needs to be added to the repository by iohyve itself (you cannot simply move an ISO to a directory).
First the creation, in this case I want to name it ubantuserver and I want the guest to have a 16GB virtual hard drive:

```
sudo iohyve create ubantuserver 16GB
```

Next we need to change some properties. I'm going to give the guest a description, change the RAM and CPU to give it more resources (2 Gig of ram and two virtual CPUs), and configure it to utilize grub-bhyve to boot the Ubuntu kernel. We can do this in one simple iohyve command:

```
sudo iohyve set ubantuserver description="Ubuntu 16.04 Server" ram=2048M cpu=2
loader=grub-bhyve os=Ubuntu
```

You can check to see if everything is set up properly with:

```
iohyve info -v
```

Be sure to put your description string between two double quotes. You don't have to set a description if you don't want to; the default description is the timestamp of when the guest was created. Next, we are going to fetch the ISO from Canonical. Since we aren't going to be using a GUI to install, we will grab the server edition from my fastest mirror (your mirror may vary, see the Ubuntu website for more info):

```
sudo iohyve fetchiso
http://mirror.pnl.gov/releases/xenial/ubuntu-16.04.2-server-amd64.iso
```

This will automatically fetch the ISO and put it where it needs to be. If you have already fetched the ISO, you can run something like:

```
sudo iohyve cpiso /full/path/to/iso.iso
```

Now here's the part where `tmux` becomes really useful. We're going to create a `tmux` window by typing Ctrl+B then "c". Ctrl+B is the default action key, similar to Ctrl+A in screen. The "c" creates the new window, pressing Ctrl+B and then "n" will cycle to the next window, and Ctrl+B and then "p" brings you to the previous window. In our new `tmux` window we are going to open a console to the new guest:

```
sudo iohyve console ubantuserver
```

Nothing should be there yet, because we haven't started the guest yet. Let's go over to our previous `tmux` window using the key command described above (Ctrl+B then C). We can see a list of ISOs that are in the local iohyve repository with:

```
iohyve isolist
```

There we should see the ISO we fetched:

```
ubuntu-16.04.2-server-amd64.iso.
```

So to begin our installation, we run the following command to boot up the new guest with the attached Ubuntu ISO:

```
sudo iohyve install ubantuserver
```

Now move on to your `tmux` window with the console open and if everything worked, you should see a GRUB menu if you're fast enough, or a wall of text scrolling by. If you have a GRUB prompt that looks like "`grub>`" then double-check your os property settings on the guest. Different distributions have different quirks that iohyve can work with. Your installation should go like any other Ubuntu installation, just be sure to choose the LVM install (which is the default). If you choose to install directly onto ext4 without LVM, you may want to set your os property to "debian" as the quirks should be similar to non-LVM Ubuntu installs. Sometimes you can get kernel messages in your console while installing, specifically during partitioning. This is fine, and just a product of using a serial console. If everything goes well, you should be prompted to reboot. Click okay and then head back over to your previous `tmux` window. If iohyve won't automatically reboot the guest, you can make sure it is no longer running and then start the guest for its first full boot:

```
iohyve list
sudo iohyve start ubantuserver
```

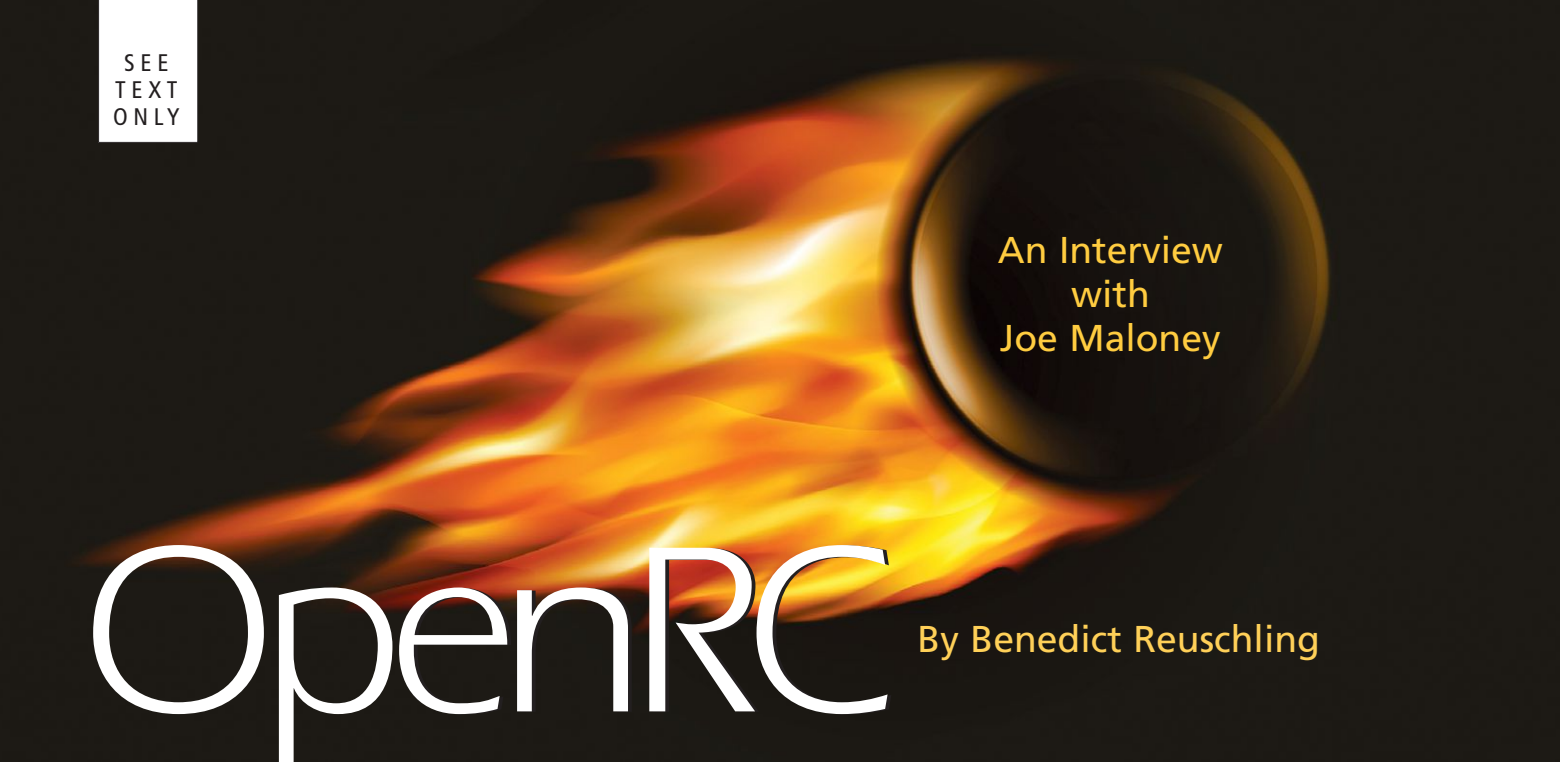
On your `tmux` window with the console open, you should start to see the new Ubuntu Guest boot up. From there you can do whatever it is you want to do with Ubuntu. If you don't like Ubuntu, take a peek at the man page to see what OSes work with iohyve's ability to manage quirks. You can rinse and repeat as long as you have the resources to do so. To keep an eye on my resources, I create a special `tmux` window with four different panes. Check the `tmux` man page for how to create and size panes, but if you are lazy like I sometimes am, you can just run these four commands in a different `tmux` window. Using panes just gives it a fancy "control panel" feel to it. First we run `htop` as root, and filter it to only view processes that start with `"bhyve"`. This will give us visibility into what bhyve guests are using how much CPU and RAM (with those fancy bar graphs!). In the next `tmux` window or pane, I like to have a current view of the installed guests and what resources they have, and whether or not they are running with iohyve info. The next two are systat outputs that give both a view of disk usage and network usage. Remember to correlate the `"tap"` interfaces in the systat output to the output of iohyve info.

```
sudo htop
iohyve info –sv
systat –iostat
systat –ifstat
```

Hopefully you now have the knowledge to maybe start moving some of those Linux servers lying around to a virtual host, or maybe just to have a nice sandbox to test new things in to help you learn a new skill. You may find that you like to monitor your resources differently, or that you only need to run FreeBSD guests, or only Windows guests. I hope that iohyve helps you navigate the system administration waters. If you have problems or questions with iohyve, or you'd like to request a new feature, or even contribute to iohyve, head on over to the GitHub page (https://github.com/pr1ntf/iohyve) and someone will eventually get to you. The iohyve project is run by volunteers, so don't expect an instant response, but we'll generally respond to a GitHub issue (https://github.com/pr1ntf/iohyve/issues) eventually.

TRENT THOMPSON is a security engineer by day and a FreeBSD and virtualization hobbyist by night, maintaining and contributing to The iohyve Project. When not doing BSD-related activities, you can find him tinkering with something else technical around the house, like musical synthesizers, model rockets, or micro-computers from the 1980s. You can never have too many hobbies.

SEE TEXT ONLY



An Interview with Joe Maloney

OpenRC

By Benedict Reuschling

Tell us a little bit about yourself, and how you got involved with OpenRC.

● My name is Joe Maloney, and I am the Lead Systems Architect for the TrueOS project. My primary responsibilities include helping with the overall direction of the project, support, and Q&A for the project. I can partially be blamed for the name change away from PC-BSD as well as the rolling release model concept that drives our development pace. I am even more to blame for the integration of OpenRC into the TrueOS project.

A couple of years ago while attending my first conference at vBSDcon 2015, I was inspired to help pursue the portability use case. Replacing `init` would not have provided the gradual approach we desired, so we landed on OpenRC, which allows us to evolve our project in phases.

Before talking about replacing the `rc` system, can you give us a brief overview of the FreeBSD boot process?

● Within the FreeBSD boot process, the kernel loads any drivers specified to run early using `loader.conf`. This means that drivers are loaded as modules before the kernel boots. The kernel itself then boots within about 6 seconds. The system `V init` program `/sbin/init` starts as `pid 1`, and executes a shell file called `/etc/rc`.

What exactly is `rc`?

● The `/etc/rc` file is nothing more than a shell script that is executed by `/sbin/init`, which exists to execute processes at bootup. Prior to `rc.d`, all processes had to be hardcoded into `/etc/rc`.

This approach had the cons of hanging the boot process if a mistake was made while the user was editing `/etc/rc`. Users would have to keep track of service ordering when using this method. They would have to daemonize processes by hand, and of course this would provide no easy means of checking service status.

What is `rc.d`?

● To solve these issues, and other problems, FreeBSD adopted `rc.d` from the NetBSD community. The `rc.d` system is a collection of shell scripts that often source other shell scripts. This slows down the system considerably when you look at something like `netif`, which, including the sourcing, processes over 4,000 lines of shell as a one-time action at bootup.

The `rc.d` system allows for arguments startup order, service status, and daemonizing within each script. Base default configuration for defaults is done in `/etc/defaults/rc.conf`. User configuration for `rc.d` is typically done with `/etc/rc.conf`. This is where a service is enabled, disabled, or additional flags for the service are set.

What were prior efforts to improve `rc.d`?

● Over many years the PC-BSD project tested DHCP vs SYNCDHCP for wireless. The difference between the two is that DHCP would background, making the boot process a little faster, and SYNCDHCP would foreground. Unfortunately, `wpa_supplicant` was unreliable for many users in our community without foregrounding with the SYNCDHCP flag, and this hung the boot process without the backgrounding functionality.

Briefly we looked at patches for `rcorder`. There were a series of patches to enable parallel startup for `rc.d`. Unfortunately, over the years, these patches were neglected, and, as many changes were made, they were no longer able to apply cleanly without a great amount of effort.

Kris Moore, the founder of PC-BSD, also previously wrote patches to add an `rc_delay` function. This would have allowed the networking, or other services delaying boot, to run delayed. However, it was suggested within reviews that parallel startup be used instead, and these commits unfortunately died in the FreeBSD reviews process as well. Still, even if that work had been accepted, it would not have been the ideal solution. For example, think about the case where login over ldap needs to happen, and network needs to be up for that. Delaying the network would only break that use case.

Why was OpenRC developed and where does it originate?

● OpenRC is an evolution of `rc.d`. It is not an `init` system. It works with the system-provided init system. Is OpenRC a drop in replacement for `rc.d`? Well, almost. Yes, there are thousands of init scripts in ports that are being converted to a simpler format with less shell. OpenRC is mostly written in C. The primary objective to using OpenRC is to be able to reduce the amount of shell by using built-in C functions, and to simplify services.

OpenRC was written by Roy Marples, a NetBSD developer. Gentoo, PacBSD, UbuntuBSD, and others use OpenRC. Work on OpenRC originally started in the Gentoo alt fork which uses the BSD kernel. When it came time for release, the Gentoo project gave Roy the blessing to release a portable version under the BSD license.

What was needed to integrate OpenRC into the TrueOS base?

● For TrueOS we removed code for Linux portability. We found this was the primary requirement for `gmake`. To integrate into our FreeBSD fork, and build with world, this had to change. Our OpenRC implementation uses BSD make now, and is packaged along with the rest of base. We added hooks to `/etc/rc, /etc/rc.shutdown`, `/etc/rc.devd`. We did not replace the scripts entirely. We just simply added the hooks to start OpenRC as the upstream project would have if installed using `gmake`. We reworked many `rc.d` scripts to be `init.d` compatible. The `/etc/init.d/` location for base scripts replaces `/etc/rc.d`. Both `/etc/defaults/rc.conf` and `/etc/rc.conf` no longer start, or stop, services but can still be used for flags. The `/etc/conf.d/` directory replaces the `/etc/conf.d/` directory, which is not as commonly used by some users.

The end result looks something like this in our fork:

Modifications to hook in OpenRC:

https://github.com/trueos/freebsd/blob/drm-next/etc/rc
https://github.com/trueos/freebsd/blob/drm-next/etc/rc.devd
https://github.com/trueos/freebsd/blob/drm-next/etc/rc.shutdown

New additions with only top level Makefiles in FreeBSD modified to build these directories:

https://github.com/trueos/freebsd/tree/drm-next/etc/init.d
https://github.com/trueos/freebsd/tree/drm-next/lib/libeinfo
https://github.com/trueos/freebsd/tree/drm-next/libexec/rc
https://github.com/trueos/freebsd/tree/drm-next/bin/rc-status
https://github.com/trueos/freebsd/tree/drm-next/sbin/openrc
https://github.com/trueos/freebsd/tree/drm-next/sbin/rc-service
https://github.com/trueos/freebsd/tree/drm-next/sbin/rc-update
https://github.com/trueos/freebsd/tree/drm-next/sbin/start-stop-daemon
https://github.com/trueos/freebsd/tree/drm-next/sbin/supervise-daemon

Were there any roadblocks during the migration?

● During the migration process, it became clear that neither parallel startup or `rc_delay` would ever have truly fixed much at all. Especially when processes foreground rather than background. I believe `rc.d` itself could be improved by simply modernizing many of the scripts, and trimming the fat where possible. The proof for me was running a smaller, and simpler network script in shell, and seeing the difference for myself.

Our first effort can be seen here:
https://github.com/trueos/freebsd/blob/fa10ac7ceb7048baad84fd3455a77edeb118c0a2/etc/init.d/network

That script works with parallel startup, and `netif` does not.

The FreeBSD `netif` script itself is fairly minimal:
https://github.com/freebsd/freebsd/blob/master/etc/rc.d/netif

However, the files it sources for functions are where the performance problems begin:
https://github.com/freebsd/freebsd/blob/master/etc/network.subr
https://github.com/freebsd/freebsd/blob/master/etc/rc.subr

What was missing was `ip alias`, `lagg`, and some other edge case functionality. Therefore it was decided to move back to `netif` for the time being for overall compatibility. This has somewhat hurt our performance, and prevented parallel from operating properly under the circumstances. However, we cannot justify keeping users from that functionality.

We can of course remove `netif` from the boot runlevel, and gain better performance by simply backgrounding networking by allowing `devd` to start it later. In TrueOS we need something to apply only specific configuration when needed rather than just a one-time evaluation at startup. Again, we are back to the issue of not really solving anything here for someone who needs network at the login manager. Some of us think that the way forward may be to write a network manager that works more efficiently in event-driven scenarios with `devd`.

Another consideration is that we currently use `dhcpcd` as our default `dhcpcd` client:
https://github.com/trueos/dhcpcd

We have not yet accomplished the task of integrating directly into the tree as NetBSD has:
http://cvsweb.netbsd.org/bsdweb.cgi/src/external/bsd/dhcpcd/

However, this is on the roadmap to complete integration for TrueOS as well as offering `dhclient` integration in base for those who wish to use it. We found in the particular case of parallelization efforts that `dhcpcd` worked better for that use case, so we chose it as our default `dhcp` client. We will have to revisit dhclient in the future.

To summarize on our to-do list:

- Import `dhcpcd` into base
- Add `dhclient` support
- Try to fix `netif` parallelization
- Possibly write a network manager down the road and make it the default option

What improvements did you gain from the migration to OpenRC?

● Even without parallel startup, TrueOS averages a 10- to 15-second startup in most cases, a 127% improvement. Some would argue they have a 9-second startup without OpenRC but obviously they are not running all of the many services we are in TrueOS out of box to support a more general use case for the average user. Of course, no login manager, and direct startup to fluxbox without `cups`, `avahi`, and many services will be faster. However, if you start 100 services, you will see a slowdown when `rc.d` is used. That is where parallel startup is useful.

We also have service supervision capability directly integrated into OpenRC. We are currently using it for sysadm to monitor the service, and restart it in the case of crashes. Other supervision backends such as `s6` can also be used even without replacement of the `init`. This can provide socket activation functionality. However, the built-in supervisor has proven functional enough for our use case.

We now have simpler service files due to `openrc-run` built-in functions. Some services can simply start with name and command specified. All services now report service status properly. With FreeBSD's `rc.d` services, we were finding that many services were not reporting service status properly. We have found the same services fixed during the migration to OpenRC.

OpenRC also provides a nice representation of the startup of services. It makes it easy to see fail-

ures, with a warning by color code. Services with errors will show up in red, services with warnings are yellow. In the case of parallel startup, blue is the general theme, and in the case of normal startup, green is the general theme. Of course, for a more traditional `rc` look, color can be turned off with a configuration parameter in `rc.conf`.

Finally we gain the ability to provide runlevels. These runlevels are not the same as Linux SysV runlevels but allow us to order services by grouping them into a runlevel.To summarize on the improvements:

- Parallel startup capability
- Service supervision capability
- Simple startup scripts
- Improved reporting of service status
- Improved display of service startup
- Runlevels, and stacked runlevels

Lastly our most recent version including OpenRC 26.2 includes the ability to show service uptime for supervised services, and we are now monitoring services.

Can you provide us with a project-hosted `init.d` script example?

● Here is an example that uses the new supervise-daemon:
https://github.com/trueos/sysadm/blob/master/src/init.d/sysadm

The `sysadm` service will use the supervise daemon to keep the service running even if it crashes. The process will only stop if the user stops the service cleanly.

What about an integrated script example for the ports tree?

● For ports the following overlay structure is used:
https://github.com/trueos/freebsd-ports/blob/trueos-master/devel/dbus/Makefile.trueos
https://github.com/trueos/freebsd-ports/blob/trueos-master/devel/dbus/files/openrc-dbus.in

The original rc.d scripts from upstream are preserved, and can still coexist on the system:
https://github.com/trueos/freebsd-ports/blob/trueos-master/devel/dbus/files/dbus.in

How difficult is it to convert an `rc.d` script to OpenRC?

● An rc.d script can be simply converted in many cases. Most of the time it just involves removing a few simple lines. A few common offenders:

- `./etc/rc.subr`
- `-rcvar=`
- `-exit_code=0-`
- `-load_rc_config ${name}`

A more in-depth guide is available on the TrueOS blog: https://www.trueos.org/blog/openrc-update-simplifying-openrc-scripts/. We also highly recommend using `man openrc-run` to view all of the available options for writing `init.d`-compatible scripts. The TrueOS project is 100% complete now with the process of adding init.d compatible scripts to ports.

Do you think FreeBSD should consider migrating to OpenRC too?

● I see FreeBSD as an excellent vehicle for appliance development. I currently hold the view that TrueOS should be seen as just another appliance based on FreeBSD.

There is more to be done but we hope to continue the work toward OpenRC to make it an even more attractive option in the future. Of course, the TrueOS project considers the OpenRC licensing model to be just right, and aside from the netif parallelization bottleneck, there is not much work to be done aside from general script conversion.

BENEDICT REUSCHLING joined the FreeBSD Project in 2009. After receiving his full documentation commit bit in 2010, he actively began mentoring other people to become FreeBSD committers. He is a proctor for the BSD Certification Group and joined the FreeBSD Foundation in 2015, where he is currently serving as vice president. Benedict has a Master of Science degree in Computer Science and is teaching a UNIX for software developers class at the University of Applied Sciences, Darmstadt, Germany.

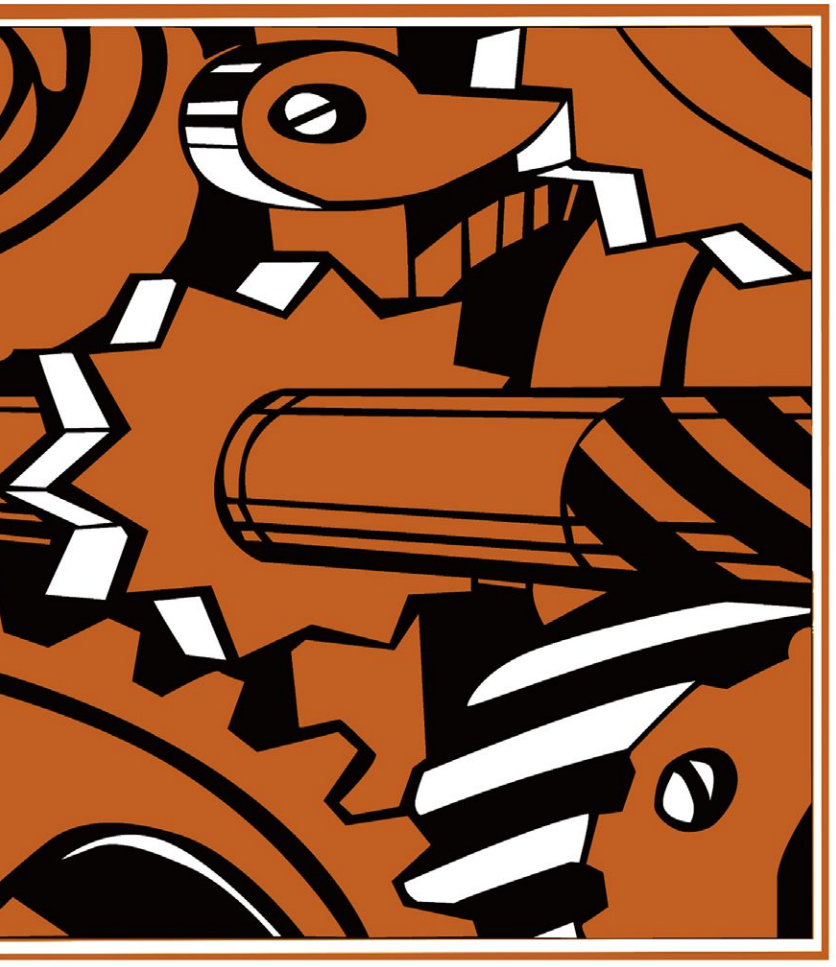
SEE TEXT ONLY

Understanding Docker

For FreeBSD Users

by Kurt Lidl

Docker, from Docker Inc., is a popular containerization software system for building, deploying, and running Linux applications. Docker containers offer a relatively low overhead mechanism for running multiple Linux applications where the different applications are isolated from each other. Docker offers a high-level interface to configure, build, store, and fetch Docker images. This article contains a brief review of popular virtualization technologies, an example of Docker's facilities to build a container, and a brief discussion of Docker's future evolution.



Overview of Virtualization Techniques

There are many different virtualization techniques available across the operating systems in use today. The most basic virtualization is the concept of a software process. This is the traditional virtualization that Unix and many other operating systems have provided to different processes from early on: each user process has an independent, protected-access memory map provided through the virtual memory system of the kernel. Other more comprehensive virtualization techniques, such as software, hypervisor-based, hypervisor-based with hardware acceleration and containerized applications, will be reviewed.

Software Virtualization/Emulation

Software-based, complete machine emulators, such as QEMU and SIMH, can emulate practically any CPU and machine architecture on the hosting machine. These types of emulators are generally fairly slow, but offer complete independence between the emulated hardware and the

hosting machine. The emulation software provides an instruction by instruction emulation of the target machine and provides a software implementation of the hardware devices of the target machine. For example, disk drives on the target are often emulated with plain files on the hosting machine. Even machines that no longer have operating hardware, such as the Honeywell DPS8M, can be emulated. In this case, the emulation is of sufficient fidelity to allow the historically significant Multics operating system to run on the emulated machine with no software changes. Another significant example of this type of emulator was the Connectix Virtual PC software, which could emulate a complete x86 computer, hosted on a PowerPC-based Mac computer. The Connectix company was purchased by Microsoft and the software is no longer available.

Hypervisor-Based Virtualization

At the opposite end of the virtualization spectrum are hypervisor-based implementations. A hypervisor-based virtualization generally runs at a significant percentage of the native speed of the hosting machine. Only a small set of hypervisor-mediated system functions execute in the hypervisor and the rest of the user code runs in the virtualized machine at native speeds. This type of virtualization is considered fairly "heavyweight," in that each virtualized machine has its own copy of whatever operating system is being run. One area of performance issues with this scheme is that the virtualized operating system also has to maintain its own set of memory protections for its own use. Hardware support for this type of operation, sometimes called "nested page tables," greatly enhances the operation of guest operating systems under the hypervisor.

There are many hypervisor-based virtualization platforms available, including:

bhyve on FreeBSD
KVM on Linux
xhyve on MacOS X
Hyper-V on Microsoft Windows
ESXi and vSphere from VMware
Xen on multiple operating systems and several hardware architectures

Containerized Virtualization

Containers are lightweight virtualization schemes where processes have some sort of partitioning and isolation between different administrative groups on the same host. The different partitions all share a single kernel application binary interface (ABI), running against a single kernel instance. Often, but not always, each process running in a container can be seen on the hosting server. This type of virtualization is generally called "container computing," and offers a middle ground between the level of isolation from hypervisors and the "shared everything" from a standard Unix environment.

Containerization is the fundamental idea behind the following facilities:

Jail system on FreeBSD
Control Groups on Linux
Containers on Nexenta OS
Containers on Solaris

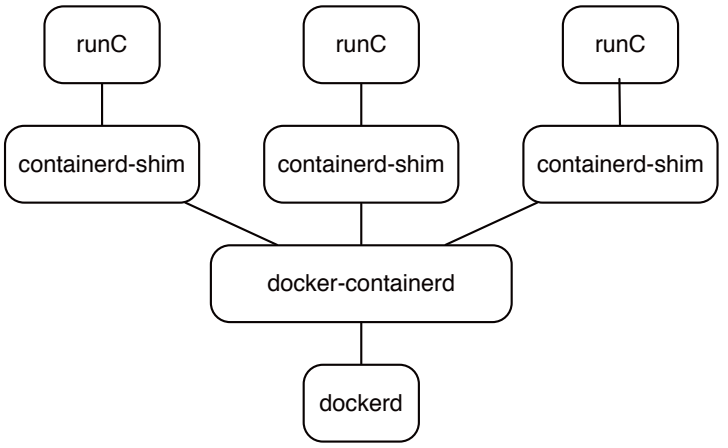
Hybrid Virtualization Techniques

There are other hybrid virtualization techniques, such as running a combination of hypervisor virtual machines and then hosting various containerized applications on those virtual machines. This hybrid approach is how Docker is implemented on non-Linux machines such as the MacOS version of Docker, which is built on top of the MacOS xhyve virtualized machine. In a similar fashion, the Windows implementation of Docker uses the Hyper-V hypervisor to create a virtual machine running Linux, which is then used to execute the system calls from the Docker containers.

Linux Control Groups and Docker

The Linux kernel has a relatively new capability that makes Docker possible: Control Groups (aka "`cgroups`"). This is the fundamental technology that allows for the isolation of various user processes in one control group from affecting and directly interacting with a different control group.

In a traditional Unix environment, there is a single hierarchy of user processes. The `init` process (pid 1) is the root of that hierarchy, and all processes can trace their ancestry back to that initial process. The `cgroups` facility in the Linux kernel allows for instantiating new hierarchies of processes that are contained entirely in the new hierarchy, and can only interact with other processes in that hierarchy.



Docker Process Relationships

The `cgroups` facility can do more than just create new process hierarchies; it can set up resource limits (for example, memory and network bandwidth) and attach these limits to the process hierarchies that are created. While management of the low-level cgroups mechanism via provided system utilities is possible, it is rather tedious. Docker provides a more convenient interface for controlling the `cgroups` mechanism at runtime, along with an easy-to-use system for building the static environments that will be executed later.

Docker uses `cgroups`, along with other Linux kernel facilities, such as `iptables`, for networking configuration and control and for a union filesystem (`UnionFS`) for isolating the container from the filesystems of the hosting machine. There is also a mechanism available to allow explicit sharing of directories between the host machine and the Docker containers. The union filesystem that Docker implements is layered on top of a Docker storage driver. The storage drivers that are available depend on the particular Linux system that is running Docker and provide varying degrees of performance and stability.

Docker Terminology and Software Architecture

An `image` is what Docker calls the containerized filesystem that has been created and loaded with the software layers that are needed for a particular application. When an `image` needs to run, a copy-on-write snapshot of the image is created, and that copy-on-write filesystem is called the `container`.

The process that starts a container is then placed into a new `cgroup` hierarchy. Any new processes spawned by the initial process in a Docker container will not be able to influence any processes outside of the running container, because all other processes will belong to different `cgroups`. This

isolation prevents any interaction or interference between two or more Docker containers running on the same physical host.

A modern Docker installation typically has at least two long-running daemons, `dockerd`, and `docker-containerd`. There is a single user command, `docker`, that takes multiple command keywords. This is similar to how many complex systems are controlled through a single dispatch command (for example, `git`, `hg`, and `rndc`). The docker command communicates through a Unix domain socket to the `dockerd` process. The `dockerd` process communicates with the `docker-containerd` process to specify the management of the containers on a system. There are other container software shims that are started for each container. The `runC` container runtime initializes and starts the container, and then hands the file descriptors for `stdin/stdout/stderr` over to `containerd-shim`, which acts as a proxy of sorts between the running `container` and `docker-containerd`. This intermediate process is required so that a restart of the `dockerd` process, and therefore, the restart of `docker-containerd`, can allow the new docker-containerd daemon to reattach to the `containerd-shim` for each currently running container.

Docker Images Explained

A Docker image is a virtual filesystem, packaged as a series of layers. Each layer in the filesystem is stacked on top of the layers underneath it. The ultimate view of the filesystem is the union of the filesystems that make up a Docker image. The layers in the image are built from the commands in a Dockerfile.

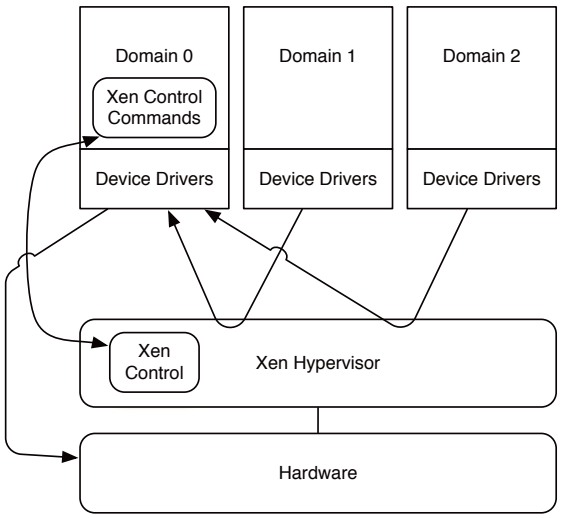
Dockerfile as a Recipe

The Dockerfile is a simple text file, holding one or more commands, and any comments that the user has placed in the file. When Docker builds an image, it runs each of the commands found in the file, in the order they are encountered. In this manner, the docker build procedure is just like following a step-by-step recipe for preparing a meal. Each of the commands in the Dockerfile will generate a new layer in the resulting image. By convention, the Docker commands are written in uppercase to help differentiate them from user-specified commands. If any of the commands that are executed fail (that is, has a non-zero exit code), the building of the image stops immediately and the image build is marked as a failure. For efficiency reasons, it is desirable to keep each layer in the Dockerfile as small as possible. This means that cleaning up after any commands that create large amounts of metadata, such as '`yum update`', should be done as part of the same command that generated the metadata.

The example Dockerfile will create an image with six layers. Some of those layers, which are identical to the prior layer, will be automatically discarded during the build process. Example: Dockerfile for running Apache

```
# An image for running Apache
FROM centos:7

MAINTAINER Ms. Nobody <nobody@example.com>

RUN yum -y --setopt=tsflags=nodocs update && \
    yum -y --setopt=tsflags=nodocs install httpd && \
    yum clean all

VOLUME ["/var/www/html", "/var/log/httpd"]

EXPOSE 80

CMD ["/usr/sbin/apachectl", "-DFOREGROUND"]
```

The first command, '`FROM centos:7`', which creates the first layer in the image, specifies that that base image for `CentOS 7` should be pulled from the central Docker repository into the local machine's cache of filesystem layers. This layer is



Xen Architecture

the bottom layer in the image. The '`FROM`' command must be the first command in a Dockerfile and initializes a new build.

The second command, '`MAINTAINER ...`', sets a special label in the metadata for the image. This label is used to identify the creator of the image. There is also a '`LABEL`' command that could be used instead, to set an arbitrary number of labels on an image. The labels can be used by the end user for any purpose.

The third command, '`RUN yum -y update ...`' updates any out-of-date software packages that were included in the base image. The next command, '`yum -y install httpd`', installs the Apache '`httpd`' package. The final part of the command, '`yum clean`', expunges all the package/repository metadata maintained by the `yum` package management system, to minimize the size of the generated layer. For the same reason, the `yum` command, using the nodocs flag, is instructed to ignore any documentation during the upgrade and installation of packages. The arguments to the '`RUN`' command are executed by a shell process, so the complexity of the generated layer in the constructed image can be quite elaborate.

The fourth command, '`VOLUME [ ... ]`', marks a list of directories to be used as external mountpoints in the UnionFS filesystem. During container execution, these mountpoints will have external filesystems mounted at these locations. The UnionFS layer should not attempt to capture that activity to the copy-on-write filesystem. This is one method how a container can persist data outside of the copy-on-write image from which the container is executing.

The fifth command, '`EXPOSE 80`', provides information that will be used when a container is started from this image. A port on the hosting machine can be mapped to the specified TCP port number of the container at runtime. Or, by specifying a different networking option at runtime, the port of the container can be made accessible to other containers running on the same host.

Finally, the sixth command, '`CMD ...`', specifies the default command to be executed when a container is started from this image. In this case, it starts the Apache webserver in the foreground. When the Apache webserver process exits, the container will be automatically stopped. It is often useful to create a small wrapper script around a daemon that is started inside a Docker container, to restart the daemon in the event that it stops running. By automatically restarting the daemon, the Docker container can continue to run without needing to be restarted.

Now that the purpose for each of the lines is known, building an image is straightforward. Note that some of the output from the build process has been removed and lines wrapped to improve readability. The image is created by running the command `docker build directory`, where `directory` is the path to the directory holding the Dockerfile (output below).

Docker Image Inspection

It is instructive to look at some of the metadata about that image, via the `docker inspect` com-

```
# docker build -t centos-apache-testimage .
Sending build context to Docker daemon  2.048kB
Step 1/6 : FROM centos:7
 ---> 3bee3060bfc8
Step 2/6 : MAINTAINER Ms. Nobody <nobody@example.com>
 ---> Using cache
 ---> 7f88dbad6a42
Step 3/6 : RUN yum -y --setopt=tsflags=nodocs update &&
            yum -y --setopt=tsflags=nodocs install httpd &&
            yum clean all
 ---> Using cache
 ---> f50595808f75
Step 4/6 : VOLUME ["/var/www/html", "/var/log/httpd"]
 ---> Running in bce2b6331fc8
 ---> 51b4c07c8eba
Removing intermediate container bce2b6331fc8
Step 5/6 : EXPOSE 80
 ---> Running in 073e6fac8709
 ---> 5bf7cadf8102
Removing intermediate container 073e6fac8709
Step 6/6 : CMD /usr/sbin/apachectl -DFOREGROUND
 ---> Running in e5a44065f0d7
 ---> 4d119d3a4776
Removing intermediate container e5a44065f0d7
Successfully built 4d119d3a4776
Successfully tagged centos-apache-testimage:latest
```

```
# docker inspect centos-apache-testimage:latest
[
    {
        "Id": "sha256:db9314a42feb [...]",
        "RepoTags": [
            "centos-apache-testimage:latest"
        ],
        "ContainerConfig": {
            "ExposedPorts": {
                "80/tcp": {}
            },
            "Env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin: [ ... ]"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ",
                "CMD [\"/usr/sbin/apachectl\" \"-DFOREGROUND\"]"
            ],
            "Volumes": {
                "["/var/www/html",": {},
                ""/var/log/httpd"]": {}
            }
        },
        "DockerVersion": "17.06.0-ce",
        "Author": "Ms. Nobody <nobody@example.com>",
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 275797466,
        "GraphDriver": {
            "Data": null,
            "Name": "aufs"
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
                "sha256:dc1e2dcd [...]",
                "sha256:41fc3fb9 [...]"
            ]
        }
    }
]
```

mand. Not all the metadata is shown in this output (above), just some of the more interesting pieces.

The "`ContainerConfig`" section has the complete environment specified for the processes in any containers that are started from this image. The "`Architecture`" and "`Os`" settings show that the containers support the Linux syscall interface, for the amd64 (aka "x86_64") machine type. The Docker image for this article was created on a Macintosh computer, running macOS Sierra 10.12.5, but any containers will be executed with a Linux/amd64 runtime environment.

This image could be moved to any host capable of running Linux/amd64 Docker images. The portability of images is one of the principle advantages of Docker ease-of-building and deploying across many different hosts, without having to worry about shared library conflicts, or corrupting configurations of already installed components. Docker supports image registries where images may be stored and retrieved. A private Docker registry can be created that allows users to centrally store their customized images. Once the image is stored in the registry, a single command can retrieve the image to a host, and a second command can start a container from that image.

Running a Container

It is easy to create a running container from the example image:

```
# docker run --rm -d -p 8080:80 \
  -v $(pwd)/htdocs:/var/www/html \
  -v $(pwd)/logs:/var/log/httpd \
  centos-apache-testimage:latest
```

This command starts the container, telling Docker to throw away the copy-on-write filesystem (`--rm`) when the container exits. Run the container in the background (`-d`) and port-map localhost:8080 to the container's tcp port 80 (-p 8080:80). Perform volume mounts of `$(pwd)/htdocs` to the DocumentRoot of the webserver (`-v $(pwd)/htdocs:/var/www/html`), and mount `$(pwd)/logs` to hold the log files from the web server running in the container (`-v $(pwd)/logs:/var/log/httpd`). Finally, the name of the image to be used as the initial filesystem for the container is given.

Looking at Running Containers

Once a container has been started, it will run until the initial process that started the container exits. The user that started the container, or system administrator, can stop the container via the `docker stop` command. This sends a `SIGTERM` to the initial process in the container, and then a `SIGKILL` after a grace period, if the container has not stopped. This is very similar in intent to running the shutdown command on a Unix host. The `docker kill` command just sends a `SIGKILL` to the root process in the container.

The `docker ps` command gives the list of running containers. The `docker rm` command can be used to remove the copy-on-write filesystem of a stopped container.

Looking at Images on the Machine

The `docker images` command will show the list of images currently available on the host.

The `docker rmi` command can be used to remove a reference to an image, freeing the storage associated with that image when the reference count drops to zero. Note that shared layers in the image may also be used by other images, so there often isn't a one-to-one correspondence between the amount of space listed as in use for the image and the amount of disk space used by the image. Many of the issues with double-counting of storage blocks that occur with filesystem snapshots are also evident with the `docker-containerd` storage of images.

Other Docker Commands

There are other Docker commands available that can be used for launching containers automatically and maintaining a set of containers that must run together to accomplish a given task. It is beyond the scope of this article to fully examine the complete set of commands available inside of Docker. More advanced orchestration of multiple containers running across multiple hosts is possible via the "`Docker Swarm`" support in recent versions of Docker.

Comparison with FreeBSD Jails

Docker containers are similar to FreeBSD jails, in terms of what virtualization is provided and how machine resources are shared. Both offer compartmentalized processes running against a single kernel image on the hosting machine. Docker offers an easy-to-use command line interface for creating, deploying, running, and updating images. Little setup and configuration are required on the hosting machine, other than the basic Docker Engine installation. FreeBSD jails have a considerably simpler interface to running and stopping jails. The base FreeBSD system offers essentially no high level support for the building and installation of jails into a directory. There are several add-on FreeBSD ports (for example, `ezjail`, `qjail`, and `qjail4`) that attempt to make jail usage less cumbersome, to varying degrees of success.

One significant advantage that Docker has over jails is the concept of spawning a per-instance copy-on-write filesystem for each container that is started. This is fundamental to the deployment and reusability of Docker images, whereas each FreeBSD jail typically runs in a persistent filesystem tree. Some of the add-on jail management systems use ZFS's snapshot and promote features to create a clone of a prototype filesystem for a newly instantiated jail, but that clone still persists the file hierarchy across multiple restarts of the jail.

With Docker containers, the Docker infrastructure takes care of mounting various directories when the container is started, whereas mounting of any directories, even including the crucial `devfs /dev mount`, must be handled explicitly for each FreeBSD jail.

For example, running the example docker container for apache, there are several mount-points active:

```
# df
Filesystem     1K-blocks      Used  Available  Use%  Mounted on
none            61890340    863500   57859916    2%  /
tmpfs            1023384         0    1023384    0%  /dev
tmpfs            1023384         0    1023384    0%  /sys/fs/cgroup
/dev/sda2       61890340    863500   57859916    2%  /etc/hosts
shm                65536         0      65536    0%  /dev/shm
osxfs          976426656 624064128  352106528   64%  /var/www/html
tmpfs            1023384         0    1023384    0%  /sys/firmware
```

The Docker system manufactured the required mounts automatically, with the exception of the `/var/www/html` mount, which was specified on the command line when the container was started.

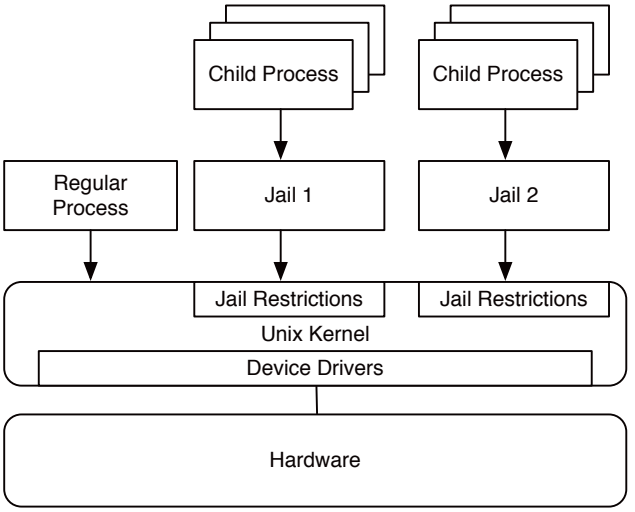
The security benefits of Docker vs jails are roughly equivalent. The security features of Docker are typically modified through command-line flags, while the security features for jails are either globally specified via sysctl settings, or have per-jail configuration settings in the `/etc/jail.conf` file. Jails, when operated with the `VIMAGE` networking option, have a per-instance network stack. This implies that each jail could have different packet filtering in place. All the running containers on a Linux-based host share a single iptables-based packet filter configuration.

The control aspects of Docker vs jails are quite different. Docker has many commands and options to allow almost all configurations to happen on the command line. FreeBSD jails rely heavily on the contents of the `/etc/jail.conf` file to specify which jails are to be run and how they are to be configured. Docker internalizes much of the configuration that is the metadata for a given Docker image. By attaching this metadata to the image, the deployment to a new host is significantly eased. FreeBSD jails have no such metadata directly attached to each jail.

In a related area, some of the FreeBSD ports for helping to manage bhyve virtualized host instances, such as `iohyve`, offer some of the same type of configuration help. These systems use ZFS properties to attach the metadata about a virtualized machine to a ZFS filesystem or ZFS zvol, which represents the filesystem for the virtualized machine. Several of the earliest versions of these management tools just used well-known names for the parameters that were to be controlled: hostname, number of cpus, amount of memory, and so forth. None offered a generic, extensible tag:value configuration file that could be attached to a ZFS filesystem, although this might have changed since the earliest attempts at supporting virtual machine metadata in this manner.

Future Directions for Docker

The Docker system is undergoing fairly rapid evolution. There is a consortium of companies that have formed the Open Container Initiative (OCI). Significant members of the OCI include Amazon, AT&T, Cisco, Docker Inc., Facebook, Google, IBM, Microsoft, Oracle, RedHat, and VMware. OCI is attempting to standardize both a container runtime system (`"runtime-spec"`) and an image specification (`"image-spec"`). As a starting point for the standardization process, Docker Inc. donated their container specification to OCI, and their runtime system (`"runC"`). Some of the members of OCI have vested interests in supporting more than just a Linux syscall ABI container, and the specifications are clear in the need to support multiple ABIs, as well as multiple operating systems hosting the container runtime. A recent development in the standardization process is Oracle's release of a open-source implementation of the oci-runtime called Railcar, which is written in Rust.



Jail Architecture

Docker on FreeBSD?

Examining the current state of the Docker system, it seems that there are no insurmountable technical impediments to making the Docker system run natively on FreeBSD. The future of Docker and the support for different ABIs across containers implies that supporting a native FreeBSD kernel ABI for the containers would be possible. Obviously, this makes deployment using Docker less of a Linux/amd64 monoculture. Currently, Docker is effectively only running the Linux ABI on amd64 hardware. The Docker community, through the OCI, has already tentatively agreed to a multi-architecture system where both Linux and Windows will be supported as first-class ABI environments, across multiple hardware platforms. This cross-system support is already available in a limited fashion for the Linux ABI on IBM's Z-System hardware, and nascent support for the arm64 architecture is available as well. It should be possible to extend this multi-ABI future to include FreeBSD. ●

KURT LIDL is a Principal Member of Technical Staff at Oracle, working on the Oracle Public Cloud build team. He started using BSD Unix with 4.2BSD, and now contributes as a Committer on the FreeBSD Project. He lives in Potomac, Maryland, with his wife and two children.

Testers, Systems Administrators, Authors, Advocates, *and of course* **Programmers** *to join any of our diverse teams.*

COME JOIN THE PROJECT THAT MAKES THE INTERNET GO!

★ DOWNLOAD OUR SOFTWARE ★

http://www.freebsd.org/where.html

★ JOIN OUR MAILING LISTS ★

http://www.freebsd.org/community/mailinglists.html?

★ ATTEND A CONFERENCE ★

vBSDCon • Sept 7–9 • Reston, VA
EuroBSDCon • Sept 21–24 • Paris, France
Ohio LinuxFest • Sept 29 & 30 • Columbus, OH

The FreeBSD Project

FreeBSD FOUNDATION

SEE TEXT ONLY

NETWORK TRAFFIC ANALYSIS

Lynx

BY PAWEL JAKUB DAWIDEK AND MILOSZ KANIEWSKI

Over the last few years, a number of TLS connections inside corporate networks increased significantly. The percentage of encrypted Internet traffic passed 50% last year and it doesn't look like this trend is going to slow down anytime soon. While it is very good to observe that encryption is getting widespread adoption, it is worth remembering that security has many faces.

In corporate networks, it is crucial for security teams to control the traffic exchanged with the outside world. Incoming traffic may contain harmful software such as viruses or ransomware. Malware designers have already started to use TLS to hide traffic that should never be detected by security software. Moreover, as it is now possible to get trusted certificates for free, assumptions such as "a padlock in the browser address field means the web-page is secure" are no longer valid. There is also a huge demand to control traffic from a local network to the Internet. It may contain confidential data that should never leave internal company infrastructure.

There is a large market of specialized tools in traffic analysis that helps network administrators provide necessary safety and guidance for network traffic control. However, IDS (Intrusion Detection Systems), IPS (Intrusion Prevention Systems), and DLP (Data Leak Prevention) tools may become powerless when the traffic they are monitoring is encrypted, e.g., with the TLS protocol. One way to deal with this problem would be to block entire TLS traffic, but, of course, that is totally impractical nowadays. The only way to get insight into it is to use a controlled MiTM technique which we describe below.

Lynx, developed by our company, is an enterprise class product which performs security-driven TLS interception. Using FreeBSD as our base system, we were able to create a product that can greatly outperform competing products from leading vendors in this market. In this article, we would like to share some of our experiences. For the last two years, we have tested and experimented with a number of approach-

es to fast packet processing. We hope that our story will be interesting for everyone interested in a network system architecture.

TLS Interception

As we have mentioned before, to properly intercept TLS connections, we have to use a controlled MiTM technique (Figure 1). First, pretending to be a destination server, we accept a TCP connection from a client. Next, we create a second TCP connection to the destination server. After that, we can start to exchange packets between client and server and, as a result, see the content of the payload. Next, we start to analyze packets sent by the client and check if there is a TLS Client Hello packet that would indicate the beginning of a TLS session. When we detect such packet, we check the content of the Server Name Indication (SNI) extension, and if it exists, we save it. This extension is used to allow servers to serve more than one certificate on a single IP address. For example, if there are two servers, example1.org and example2.org, and they both are hosted on the same IP address, then SNI allows the server to choose which certificate should be presented to the client. It is also possible that server will have one certificate that matches both SNIs. However, this is not required and a server may have two certificates, as in our example.

After reading the SNI, we suspend communication with the client and we establish a new TLS session with the server (of course using same SNI that we have stored just before). After establishing this TLS connection, we know what the server certificate looks like. Having that knowledge, we can next generate an almost identical certificate and sign it using the custom certificate authority. We can then resume communication with the client, complete the TLS handshake and present the just generated certificate. If we want the TLS session to be successful, the client needs to trust this certificate. Usually, in corporate networks, TLS clients (e.g. web browsers) are already configured to trust the additional CA certificate used to sign certificates of internal resources. If Lynx uses the same CA (or sub-CA) to sign certificates that it generates, then the client will automatically trust them.

This step finishes the negotiation phase and leaves us with two TLS channels: one with the client, and one with the server.

As we can see for every intercepted TLS session, Lynx has to handle two independent TLS connections: one with a client and one with a server. Being a part of both connections, Lynx has the ability to see decrypted traffic transmitted in two directions. This decrypted data can be next sent to other tools that can analyze its content looking for malware or data leaks.

Beginning of the Lynx Architecture

The very first version of the architecture we experimented with used one process per connection. Of course, that is a very naive approach performance-wise, but has some great security properties—we could close each process in a very tight sandbox using Capsicum, thus isolating every connection from one other. With such an approach, we could handle a relatively small number of connections: we were limited by a number of processes in the system, which, by default, is 100,000, and we needed a lot of memory. We also put huge pressure on the scheduler to switch between all those processes as the packets were arriving.

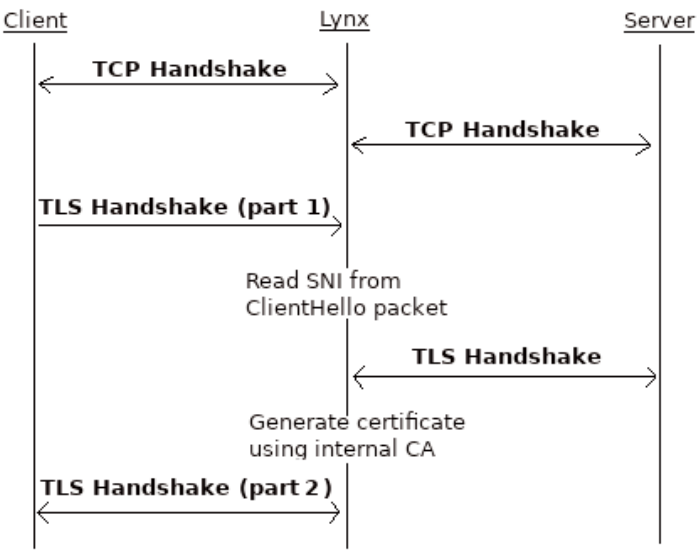


Fig. 1

We even didn't try to optimize this architecture, as it didn't allowed us to handle hundreds of thousands of simultaneous connections which was already our goal. Using threads

was also not an option, as it resulted in the same problems (mostly a scheduler overhead). Instead we moved to a fully non-blocking and event-driven architecture. This approach to packets processing was, at this time, already implemented in some network applications like Nginx. As the name suggests, it is based on two main concepts:

Non-blocking:

In non-blocking programming, a user program never performs any operation that would block its execution. If some resource is not immediately available, then the function returns and it is the responsibility of the user to call this function again later.

Event-driven:

The user program needs to know when it should repeat a non-blocking operation. In the classical approach, the user program regularly polls a specified resource to check if its state has changed. However, polling may be expensive. Generally a better solution is to let the operating system inform the user program about resource availability. This kind of information is called an event.

The result of this approach is a finite and small number of worker processes. All workers are listening on the same accept socket and it is the responsibility of the kernel to share incoming connections fairly between them. By binding workers to particular CPUs, we guarantee that scheduler overhead will be completely unnoticeable.

To adapt Lynx to this new model, we had to localize all blocking operations and rewrite them to non-blocking versions. First, we needed to modify all network I/O operations, which wasn't too hard, as non-blocking operations are supported by BSD sockets for a very long time. Also, OpenSSL modification was very straightforward. One of the resources that we had to treat specially was the PostgreSQL database. Due to the nature of DBMS, the database can execute only a single query at one given time. One way to execute more than one query is to have multiple database connections. However, their number is limited (in the case of PostgreSQL, it should not be bigger than several hundred), and using too many of them may affect performance. Therefore, the only way to handle more queries is to queue them at the level of the user program. To give workers the illusion that they communicate with the database in a fully non-blocking manner, we decided to delegate direct database operations to a separate module. To make communication between a worker and a module non-blocking, we have prepared a modified version of an nv(3) interface (we have named this new IPC "nvlnbio"). We also started using separated modules in all situations where workers needed to communicate with some resource that did not offer a fully non-blocking API (Figure 2).

Some time later during the tests, we discovered that in the case of a large number of TCP connections per second, our worker model does not scale very well. It turned out that it didn't scale linearly—every additional worker resulted in worse "per-worker" performance, and after four workers, we stopped scaling at all, even though each worker had a dedicated CPU core (Figure 3).

To investigate this problem we decided to look for locks used by the network stack during heavy

Fig. 2

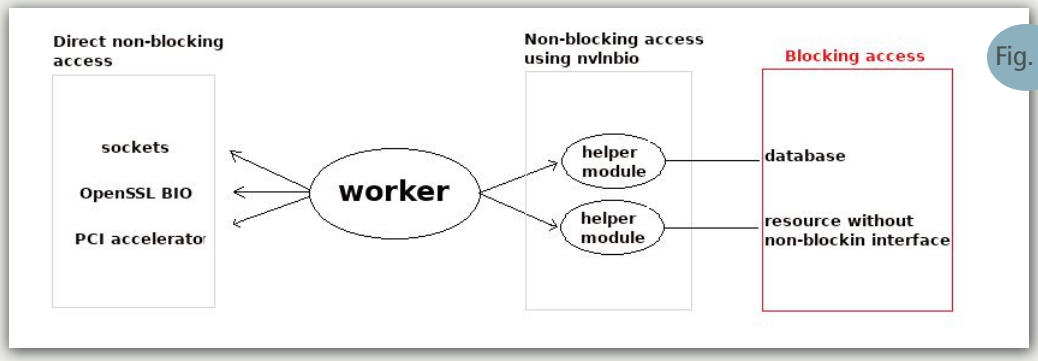
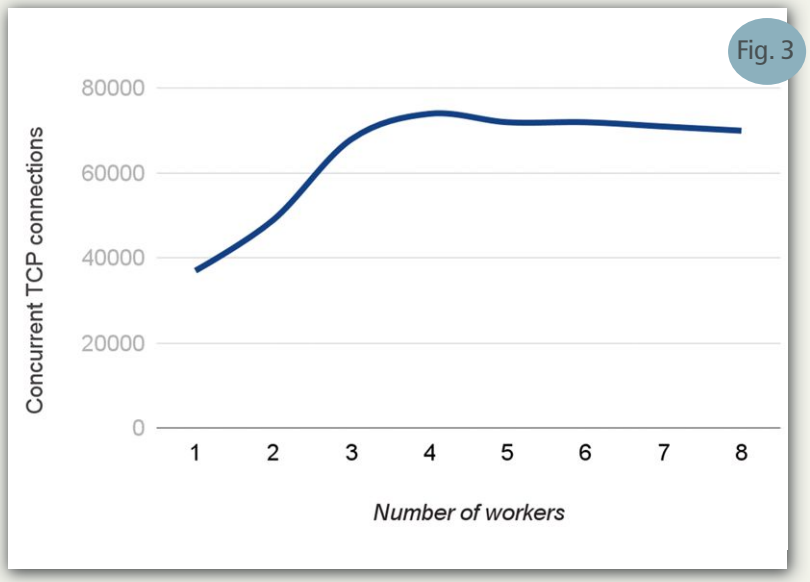


Fig. 3



load (we were using FreeBSD 10.2). For this purpose we used a LOCK_PROFILING(9) interface provided by the kernel.

First we checked lock usage with one worker running (Figure 4).

Next we repeated the same situation, but with two simultaneously running workers (Figure 5).

When two workers were running, the new type of locks related to TCP processing started to be hot. When more than two workers were launched, those locks became more and more active, and we suspected that they may be a source of the performance decrease we observed.

We had also used dtrace to detect all situations in which blocking could occur. Our goal was to have a program that blocks only if there is no work to do. We wanted to eliminate blocking on socket operations, IPC communication or reading data from the disk. The only place that we expected to be blocking was on invocation of the kevent(2) function on the main event loop. Therefore, we created a dtrace script that informs us whenever blocking occurs in any other function (next page).

The sleepq_add(9) is the best place to check if the process is blocking somewhere in the kernel.

max	wait_max	total	wait_total	count	avg	wait_avg	cnt_hold	cnt_lock	name
18	68	455629	2835614	3157952	0	0	0	515451	/src/sys/netinet/in_pcb.c:1618 (sleep mutex:pcbgroup)
19	74	580341	33734	8447282	0	0	0	101461	/src/sys/kern/kern_timeout.c:535 (spin mutex:callout)
2	3198	202	31632	1410	0	22	0	14	/src/sys/kern/kern_exit.c:469 (sx:allproc)
0	30284	0	30284	4	0	7571	0	1	/src/sys/dev/coretemp/coretemp.c:327 (spin mutex:sched lock 0)
60	38	699727	17510	5413739	0	0	0	48641	/src/sys/net/route.c:439 (rw:rtentry)

Fig. 4

max	wait_max	total	wait_total	count	avg	wait_avg	cnt_hold	cnt_lock	name
18	43	572746	3814587	4077828	0	0	0	572552	/src/sys/netinet/in_pcb.c:1618 (sleep mutex:pcbgroup)
310	573	1046182	3237040	582523	17	5	0	196106	/src/sys/netinet/tcp_usrreq.c:1142 (rw:pcbinfohash)
552	538	10662226	2522505	582523	18	4	0	241451	/src/sys/netinet/tcp_usrreq.c:296 (rw:pcbinfohash)
17	103	81477	1970702	1137267	0	1	0	250167	/src/sys/netinet/in_pcb.c:1322 (rw:pcbinfohash)

Fig. 5

```
#!/usr/sbin/dtrace -s

  syscall::kevent:entry
  /execname == "lynxd"/
  {
      self->inkevent = 1;
  }

  fbt::sleepq_add:entry
  /!self->inkevent && execname == "lynxd"/
  {
      printf("%s(%d)\n", execname, pid);
      stack();
      ustack();
  }

  syscall::kevent:return
  /execname == "lynxd"/
  {
      self->inkevent = 0;
  }
```

Network Stack Virtualization (VNET)

To overcome locking problems, we decided to give VNET(9) a try. We hoped that locks detected during our test would be virtualized, and as a result workers would no longer need to compete for them. The test results showed that our assumptions were right (Figure 6).

After using VNETs, hot locks on the TCP stack were no longer present. Performance tests also showed that worker scalability after using VNETs gets much better.

To easily adapt VNET technology into our architecture, we had to make a couple of changes in the kernel code. By design, VNETs are tightly integrated with jails. For example, when an interface is attached to the VNET, it disappears from the global system view and is only accessible from within the appropriate jail. Therefore, to use interfaces attached to VNETs, we would have to launch our worker programs in many separate jails, but we didn't want that. Instead, we have modified the

Fig. 6

max	wait_max	total	wait_total	count	avg	wait_avg	cnt_hold	cnt_lock	name
19	135	700557	3763943	4556117	0	0	0	480993	/src/sys/netinet/in_pcb.c:1618 (sleep mutex:pcbgroup)
19	56	1067986	93090	12345133	0	0	0	347169	/src/sys/kern/kern_timeout.c:535 (spin mutex:callout)
33	19	1169609	39617	7805677	0	0	0	95923	/src/sys/net/route.c:439 (rw:rtentry)
25	30	220474	32546	1952683	0	0	0	86194	/src/sys/kern/kern_event.c:1895 (sleep mutex:kqueue)
0	3253	251	18068	1664	0	10	0	7	/src/sys/kern/kern_exit.c:469 (sx:allproc)

socket(2) interface to allow us to define VNET numbers during socket creation (Figure 7):

```
/* Create socket bounded to VNET (jail) number 100. */
int sock = socket(PF_INET, SOCK_STREAM | VNET_TO_SOCKTYPE(100));
```

Fig. 7

Choosing VNET technology allows us to resolve one other problem that we were facing at the time. As mentioned before, Lynx only decrypts traffic and doesn't do any kind of traffic analysis. Other monitoring devices (e.g., IDS or IPS) are used for this purpose. Such devices can usually operate in two modes. The first is when they fetch a copy of the traffic and analyze it without accessing the main data path—this mode is called "out-of-band." In the second mode, a monitoring device needs to have access to the real traffic and thus to the main data path—this mode is called "inline." The out-of-band mode was supported in Lynx from the beginning, but the inline mode was still to be implemented. As Lynx itself stands in the main data path, there was a need to create a kind of sub-path to which a monitoring device could be connected. From the hardware view, such a sub-path would be just a wire loop between two NIC interfaces. The question we were facing was how we should create a TCP connection on such a loop. One possibility was to craft all TCP packets in our code (we were already doing it for out-of-band mode). However in out-of-band mode, we only send packets, so there is no possibility that TCP problems like packets reordering or packet drops would happen. In the inline mode such situations could take place (e.g., the monitoring device would drop packets) and so it was mandatory to handle them properly. Therefore, we quickly became convinced that the only possible solution was to use a full TCP stack implementation.

The most obvious solution implied usage of the FreeBSD kernel network stack. The plan was to call accept(2) on one NIC port, connect(2) on the second port and as a result establish a TCP connection where packets would appear on the wire linking two interfaces. However, it turned out that in such a situation, the network stack detects that it controls both sides of the transmission and therefore forwards all the traffic through the kernel. As a result, the TCP connection is properly established, but no packets are ever transferred over the physical wire (Figure.8).

Using VNETs solved this problem because it put two NIC interfaces into two separate virtualized stacks that now did not see one other (Figure 9). The result was that packets were forced to be transferred over the wire. VNETs also allow us to use the same TCP/IP addressing on multiple interfaces, which is not possible with a single TCP/IP stack.

Full Scalability

Up to this point, we have been talking about performance mostly in terms of the maximum number of TCP connections per second. However, another very important benchmark in the case of network devices is a traffic throughput. The nice thing about using modern CPUs is that there are no major differences between transmitting plain and encrypted traffic. With support for AES-NI acceleration, processing of encrypted packets is almost free. One CPU core (tests were performed on Intel Xeon E5-2697A v4) can process about 10.4 gigabits of encrypted data per second. This means that in theory, two cores are enough to decrypt/encrypt TLS traffic on a full-duplex 10Gb NIC port. During our test it turned out that logical cores that were available when Hyper-Threading was enabled were per-

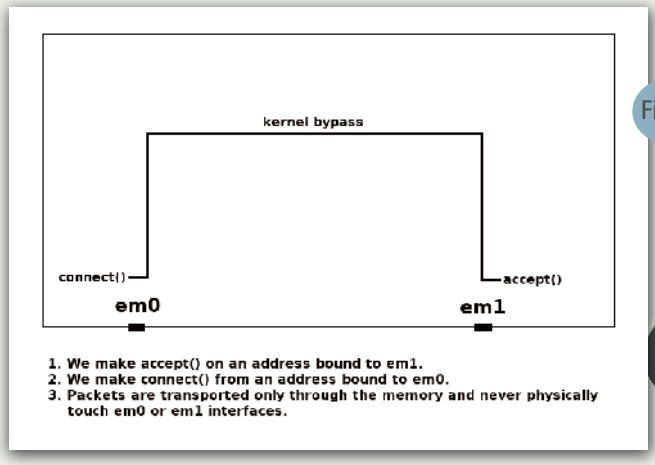


Fig. 8

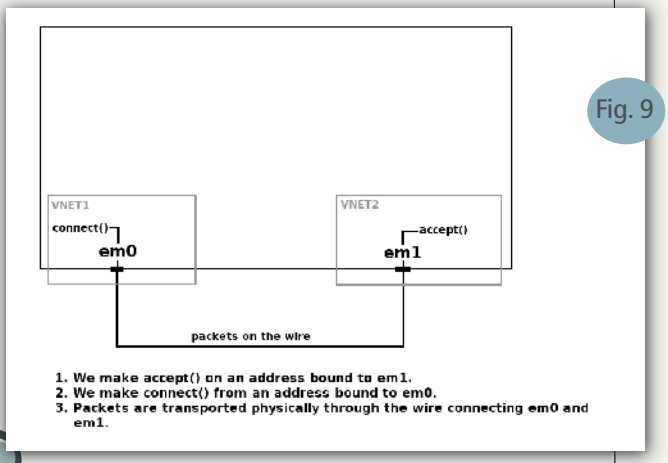


Fig. 9

forming AES as well as physical cores.

However at such high throughput, other factors start to have significant impact on performance. Most of them are related to memory:

- number of cache misses,
- memory locality in case of NUMA architecture,
- speed of RAM memory (both throughput and latency),
- number of memory channels on the motherboard,
- speed of the QPI bus.

A general approach to handling all of these cases is to guarantee that data processed by the device is read/write as infrequently as possible, and when it is happening, then as many operations as possible are performed on it (run-to-completion architecture). In other words, we need to have full control of the place where packets are processed (i.e. on which CPU) and when it happens.

The first place where we can control packets is the network card. In modern NICs, packets are processed in separate queues, and it is possible to steer how they are distributed across those queues. Usually RSS (Receive Side Scaling) is used, and packets are classified by IP addresses (and sometimes by port numbers). As a result, packets from one connection are always processed by the same NIC queue. If we want the connection in both directions to be hashed to the same RSS queue, then we need to be sure that the RSS key used for hashing is symmetric. Such a key can be easily found on the Internet.

Going forward, we would like to be able to process packets from one queue always on the same CPU core. To achieve this we have to bind all workers and corresponding IRQ threads (one per receive NIC queue) to specified cores.

All of this hardware optimization is possible using the kernel stack. However, even using VNETs, there is still a possibility that multiple workers would compete for some locks. Switching between kernel and userspace context may also decrease performance. Every read-and-write operation causes execution of a syscall which triggers a copy of memory from a user buffer to a socket buffer. To mitigate those problems, we have decided to try a solution that is recently getting widespread attention—user space TCP stack. There are many implementations of such stacks available, but we decided to try libuinet developed by Patrick Kelsey. It is a user space implementation of a FreeBSD kernel stack and, therefore, it inherits most of its attributes like stability and efficiency. It works on top of netmap, which is another excellent software that allows user space apps to get direct access to NIC buffers. Both of these solutions guarantee that after receiving a packet on a NIC it will be delivered directly to a user space program without the need to use kernel context (besides IRQ threads for communication) and with as few as possible memory operations.

Summarizing, at this point incoming packets would be processed in this order:

1. Received on NIC, which forwards it to an appropriate queue.
2. DMA packets to a memory region mapped by netmap.
3. Packets processing by libuinet stack.
4. Read packet data into worker code.

All work required in this path would be done on the same CPU core. The output route would look very similar (in our case one worker process is responsible for both receiving and sending packets, so output processing would still take place on the same CPU core).

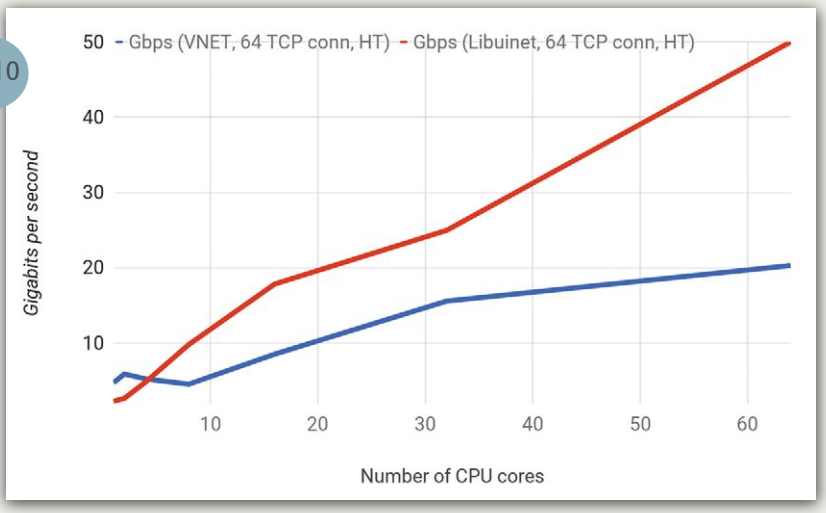
Using libuinet and netmap also allow us to opti-

mize a couple of things. We could, for example, do zero-copy of packets other than TCP, as they don't contain TLS and, therefore, are not interesting to us. This feature is already implemented in libuinet, so using it is as simple as setting an L2 bridge.

After implementing all those pieces of new architecture, we began performance tests. As we expected, the new architecture scales much better compared to the previous (VNET) solution (Figure 10)

To generate TCP traffic, we were using our own software, which worked similarly to the iperf utility. We have a client machine and a server machine that exchange random data using 64 TCP connections. Between those two machines, we put Lynx, which exchanges data between connections on both sides.

Fig. 10



Lynx is equipped with the following hardware:

- 2 x Intel Xeon E5-2697A v4 CPU (16 physical cores with HT enabled). In total, it gives us 64 logical CPU cores.
- 256 GB of 2400 RDIMM RAM placed in 8 channels (4 channels per CPU socket).
- 10 Gbps T540-CR Chelsio NIC cards.

We run our tests by increasing the number of CPU cores processing packets starting from 1 and going to 64. For one CPU, VIMAGE architecture gave us very good result (about 5 Gbps). With 4 cores processing packets, we have seen similar results for both VIMAGE and libuinet architecture. But for a higher number of cores, libuinet architecture showed its potential. While its performance didn't scale linearly, it still offered a very good ratio between processed traffic and number of cores used. With a maximum number of 64 cores, it allows Lynx to process over 50 Gbps of traffic (this is still related to TCP traffic, but with some other optimizations not described in this article, we were also able to process 50 Gbps of TLS traffic). In the same time, VIMAGE architecture lets us process only 20 Gbps of traffic.

What's Next

While libuinet architecture did a great job, there are still a lot of possible optimizations that we would like to explore in the future. We are sure that our biggest current bottleneck is the memory bandwidth. Using PCM tools, we were able to measure that the saturation of RAM throughput almost reaches its limit. We need to investigate whether all packet copying operations are required and to look for solutions that would increase the possibility that processed data is already in the CPU cache.

During our test we didn't have NUMA or COD enabled. Both of these technologies should help us achieve better memory locality per CPU socket. Without it, both processors use each other's RAM banks, which causes increase of memory access latency. Furthermore, it causes high utilization of the QPI bus whose throughput is also limited.

There is also a broad range of optimizations that can be done on NIC cards. We use Chelsio cards that offer a lot of interesting features. For example, they allow movement of packets between the ports of one NIC card without using system RAM at all. Such solutions may be used to move packets other than TCP, as they are not interesting for Lynx in any case. Also, Direct Cache Access (DCA) can reduce number of read operations and as a result, decrease RAM usage. The next stop for Lynx is 100 Gbps. Stay tuned! •

PAWEL JAKUB DAWIDEK is cofounder and CTO at Wheel Systems and the main architect of the company's products. He is also a longtime FreeBSD committer and the author of GELI disk encryption, HAST highly available storage, auditdistd, the ZFS filesystem port and many contributions to Capsicum and GEOM.

MILOSZ KANIEWSKI is a programmer and FreeBSD user. He works for Wheel Systems and is responsible for development of the Lynx SSL/TLS Decryptor. He is mostly interested in network programming, especially in software utilizing the TLS protocol.

SEE TEXT ONLY

conference REPORT™

by Roller Angel

BSDCan 2017

Prior to the first day of the Devsummit, I traveled to Ottawa, Canada, by plane from Colorado. After dropping off my luggage, I headed straight over to the Goat BoF which took place at The Royal Oak. There were already a number of people engaged in conversation over food and drink. The Royal Oak was only a few streets away from where I stayed and it was nice to walk to the conference each day, the pubs, and the grocery store. I was delighted that the people sitting with me were also into the BSDs and were happy to talk about them the whole time. I felt right at home from the start as people were very nice and were interested in what I was working on. I honestly didn't know that I would fit in so well.

I had a preconceived notion that people could be a bit hard to approach, as some are famous and so technically advanced. At first, people seemed to only be working in smaller circles. But once I got more familiar with the faces, I realized that the circles didn't always contain the same people and that they are "just people" talking about specific topics. I found it easy to participate in the conversation and also found that people were happy to get feedback. I volunteered to help wherever possible and got to work on the video crew that recorded the audio and slides of the talks. It's nice to feel welcome and a part of the community. Dan Langille mentioned in his welcome on the first day of BSDCan that the conference is like home for many.

Devsummit: Day One

On the morning of the first day of the Devsummit, I arrived early to collect my name badge and find a place to sit. Gordon Tetlow greeted me and helped me find my badge. I recognized Dru Lavigne, and so I walked over to introduce myself. Dru introduced me to Warren, who gave me a FreeBSD dog food sticker as a reward for eating my own dog food by running FreeBSD on my laptop. I introduced myself to Allan Jude, Benedict Reuschling, Sean Webb and Kirk McKusick. Shortly after, Gordon started with an introduction and information about the events of the day. There was some discussion about a new code of conduct by Benno Rice, who mentioned that people are welcome to join a group that is forming to help work out issues related to code of conduct. Allan introduced the idea of creating a process for formally discussing big project changes or similar discussions that are going to be known as FCP or FreeBSD Community Proposal. In Python, we have the Python Enhancement Proposal or PEP which is very similar to the idea of FCP.

There was some discussion about taking non-code contributions from people and how to recognize those people in the project. There was a sug-

"It was a great experience to meet the community and participate in discussions. I'm grateful that I was able to attend my first BSDCan." —*Roller Angel*

gestion to have a FreeBSD member status that could be given to people whose non-code contributions are valuable to the project. This seemed to be on a lot of people's minds as something that should be put in place soon. The junior jobs on the FreeBSD Wiki were also mentioned as a great place to look for ideas on how to get involved in contributing to FreeBSD. During the break, Benedict talked to me about being new to the community and how I'm welcome to join the conversations and share my ideas. After the break, Intel had a representative speak about their Quick Assist Driver technology and how they have implemented it on FreeBSD. While getting lunch, I met Pierre Pronchery and ended up going with him to the hacker lounge.

Pierre offered to work with me on my security skills as he is a security consultant by trade. While in the hacker lounge I worked with Pierre on joining the EdgeBSD project as a systems administrator. We began talking about rebuilding the edgebsd.org website using Flask, as I've been studying Python web development this past year and thought that it would be great to get some experience working on a real project using Python. Pierre helped me get my PGP and SSH keys created so I could securely authenticate to the virtual machines we would be working on. Pierre then showed me how to communicate via an old program called talk. It lets you communicate with other users who are logged into the same machine as you via each other's terminals. Talk is pretty awesome! I plan on using one of the recent articles from *The FreeBSD Journal* to help get Salt working with the new website infrastructure based on Flask. Before I left the hacker lounge I spoke with Michael Dexter; he let me know that the documentation team could really use my help and that there is plenty of mentorship available to help me. I let Michael know that I planned on attending the doc lounge to get up to speed on using the tools to edit the documentation and be able to help out.

Devsummit Day Two

On day two of the Devsummit, Ed Maste spoke about the items we have, need, and want in the upcoming release of FreeBSD 12. (There was discussion about when to stop supporting the SPARC infrastructure that was pretty entertaining.) There really is a lot of planning that goes into a release. We walked to a staircase at University of Ottawa and took a few group photos. Afterwards there was a mentorship and orientation session for newbies to be able to meet new people and to find out who else is interested in similar things so that they can talk with one another, and work together. After attending the newbie session, I helped with packing the conference registration materials into the FreeBSD handbags. The handbags were then transferred from the University to the Red Lion, where I also did some training on the gear for recording the talks. Before the training, I sat with Peter Hansteen and talked about using JavaScript for writing slides for talks and how that approach makes sense although it does spark some controversy. Once I finished with the training, I visited the doc lounge and began getting my computer set up for editing documentation.

BSDCan: Day One

On the first day of BSDCan, I arrived early to coordinate with the team that records the talks. We selected the rooms where each of us would do the recording, and set up a group chat via WhatsApp for coordination. When the conference began, we heard from Dan Langille in the welcome session. The keynote address was by Professor Michael Geist of the Canada Research Chair in Internet and E-commerce Law. He focused on the reality that the legal industry needs to hear the voices of the technical community, and also highlighted 10 major issues in law and technology. One of the issues is access—all should have access to decent Internet speed to ensure better competition by allowing the smaller players to use roaming services on the larger networks. He concluded by stating that the legal industry desperately needs ideas, expertise, and more voices speaking out. After Michael answered a number of questions from the audience, we branched out into the different rooms to begin the talks.

I recorded Michael Tuexen's talk about packetdrill which was very interesting and the only talk in the FreeBSD Developers Summit track. After that, I attended the talk about FreeBSD on ARM by Emmanuel Vadot. The poudriere image BoF was next (in the same room), which seemed to be a great fit as the discussion was already on embedded devices and naturally evolved into discussion about creating the device images. I asked about current support for building images for the MIPS architecture boards. I found out that the support isn't in poudriere at the moment and is being worked on under the FreeBSD WiFi build project. After that, I stayed in the same room for the talk on the realities of DTrace on FreeBSD by Brian Kidney. The talk went great, there was a lot of information on the state of DTrace and there was a good amount of participation and questions from the audience. After taking a break to get some much-needed rest, I went to the doc lounge to sharpen my skills. Tim Moore was there and helped me get my text editor set up to

make it easier to edit documentation. I found out how to edit and submit documentation patches in no time. Dru was sitting right next to me answering any questions I had as I went through the process.

After the doc lounge, I visited the hacker lounge. There were already several tables full of people talking and working on various projects. In fact, there was a larger, jolly group of people who were collaborating on the new libtrue library. I did a little socializing and then got on my laptop and did some more work on the documentation using my new skills. I really enjoyed having a hacker lounge to go to at night.

BSDCan: Day Two

On day two of the conference, I arrived early to help the video crew extract the previous day's recordings from the laptops and get the laptops set up in each room for the last day of talks. The first talk I recorded was Ken Moore's talk on SysAdm. Ken shared the progress made on the project and discussed the various features and how they were implemented. Next, I recorded the session on Hardening Pkgsrc by Pierre Pronchery. I didn't realize that pkgsrc can be secured in multiple ways without too much pain and that it will actually work on any BSD you would like. Using tips from this talk, I plan on digging deeper into pkgsrc in the future. After the morning sessions, I grabbed some lunch and quickly went to the BSD User Group BoF. There we had great discussions of past BUGs and how they were successful, current BUGs and how they're operating, as well as ideas for those starting a new BUG. I talked about my experiences with Colorado BSD Users Group (CoBUG) and Front Range BSD Users Group (FRBUG). I also asked for feedback on my interest in starting a Python BSD Users Group, a group for people using Python on the BSDs. I was told to just give it a shot and see if there's any interest, and that may be exactly what I end up doing as I find the time. After the BoF, the video crew had enough people recording talks, and so I decided to attend the talk on building high-performance websites with FreeBSD in the cloud by Kylie Liang from Microsoft. A live demo of Microsoft Azure was a large part of the session. It was strange to see someone from Microsoft using Windows to deploy FreeBSD, but it worked out great and seemed simple enough to use.

The next talk I attended was given by Stephen Herwig, a PhD student working in the systems and networking lab of the University of Maryland. He had developed a new security model for NetBSD, similar to OpenBSD's pledge and FreeBSD's capsicum, and called it secmodel_sandbox. This was a very technical talk, but one that I found quite interesting. There were several hard questions asked by the audience and Stephen was very detailed and thorough in his responses. The last talk of the day that I attended was by Rodney Grimes, and it was about challenges and opportunities to better use FreeBSD in enterprise environments. Building an open-source operating system is a lot of work, and the relationship between the volunteers of the FreeBSD Project and the vendors is very important. The talk covered a lot of previous experiences and suggested ways that we might be able to take advantage of current trends.

After the talk, the closing session began. There was an exciting announcement of a new conference in Taiwan, http://bsdtw.org. There was also a surprisingly entertaining charity auction. Once all the goods had been auctioned off, everyone headed to the closing social event at The Red Lion. I had my picture taken with Henning Brauer wearing an I love FreeBSD t-shirt for donating $10 to charity. I also kept a running list of the names of people that I had conversations with during the conference, not counting those conversations that took place at the closing social event, as there were too many and I wasn't keeping track at that point. There are over 30 people on my list, including all three of the Moore brothers (the Moore Dynasty), Allan Jude, Michael W. Lucas, Benedict Reuschling, Kirk McKusick, Reyk Floeter, Pierre Pronchery, Sean Webb, Peter Hessler, Michael Dexter, Aaron Poffenberger, Dan Langille, Gordon Tetlow, and others.

I want to extend a big thank you to the FreeBSD Foundation for approving my travel grant. It was a great experience to meet the community and participate in discussions. I'm grateful that I was able to attend my first BSDCan. After visiting the doc lounge a few times, I managed to get comfortable using the tools required to edit the documentation. By the end of the conference, I had submitted two documentation patches to the FreeBSD Bugzilla with several patches still in progress. Prior to the conference, I expected I would be spending a lot of time working on my Onion Omega and Edge Router Lite projects, but I actually found that there was always something fun going on that I would rather do or work on. I can always work on those projects at home anyway. I had a good time with the FreeBSD community and look forward to continuing to work with them by editing the documentation and working with Bugzilla. ●

ROLLER ANGEL is a help desk technician and assistant systems administrator with The GLOBE Program in Boulder, CO. He enjoys longboarding, spending time on the computer learning, and hanging out with his family and four small dogs.

THE INTERNET NEEDS YOU

GET CERTIFIED AND GET IN THERE!

Go to the next level with  **BSD CERTIFICATION**

Getting the most out of BSD operating systems requires a serious level of knowledge and expertise

NEED AN EDGE?

BSD Certification can make all the difference.

Today's Internet is complex. Companies need individuals with proven skills to work on some of the most advanced systems on the Net. With BSD Certification

YOU'LL HAVE WHAT IT TAKES!

SHOW YOUR STUFF!

Your commitment and dedication to achieving the **BSD ASSOCIATE CERTIFICATION** can bring you to the attention of companies that need your skills.

BSDCERTIFICATION.ORG

Providing psychometrically valid, globally affordable exams in BSD Systems Administration

SEE TEXT ONLY

svn UPDATE

by Steven Kreuzer

Since I began writing this column, it has always been focused on documenting the new and exciting changes that have been recently added to FreeBSD. For this installment, I decided to shift the focus and highlight some of the features that have been recently removed. The FreeBSD Project has been around for a long time (over 20 years), and when FreeBSD 1.0 was released back in November 1993, most machines ran at 33 Mhz and had 16 megabytes of RAM. From that time, right up until today, we have seen countless new technologies become mainstream and slowly fade away into obscurity as they get replaced with something faster, cheaper, and more reliable. All technology has a shelf life, and for all of it, the user base will eventually drop to zero or the hardware will become more difficult to find. While FreeBSD developers attempt to support these systems for as long as possible, at some point, the time and effort required to do so stops making sense. Today we pay homage to the technology of yesteryear that has sadly outlived its usefulness.

Remove pc98 support (https://svnweb.freebsd.org/changeset/base/312910).

The NEC PC-9801, which was first released in October 1982, was the Japanese equivalent of the IBM personal computer. While the system was based around Intel 8086 and compatible processors and it was possible to run localized versions of Windows and DOS, the design is proprietary and not compatible with the IBM PC or clones. PC-98 dominated the Japanese computer market selling in excess of 18 million units by 1999, but when Windows 95 was released with the ability for IBM-compatible PCs to display Japanese text, the popularity of the platform started to decline.

Remove System V Release 4 binary compatibility support (https://svnweb.freebsd.org/changeset/base/314373).

UNIX System V, developed by AT&T and first released in 1983, is one of the first commercial versions of the Unix operating system. System V Release 4 (also known as SRV4), which was released in 1988, was the most commercially successful version. Throughout the 1990s, a variety of SVR4 versions of Unix were available commercially for the x86 PC platform, but with the increase in popularity of both Linux and BSD Unix, the market for commercial versions of Unix on desktop PCs quickly declined. By the early 2000s, SRV4 had pretty much ceased to exist. FreeBSD has kept SRV4 binary compatibility support, but the discovery that socket layer compatibility has been broken for quite some time made it clear that there is a good chance that no one is making use of this feature anymore.

Remove Micro Channel Architecture support (https://svnweb.freebsd.org/changeset/base/313783).

Introduced by IBM in 1987, Micro Channel Architecture was a proprietary 16- or 32-bit parallel bus that superseded the ISA bus. It was mostly used on PS/2 computers until the mid-1990s, when it was eventually replaced by PCI. Of the commonly available machines, only a few 486 machines used it, and those haven't had enough memory to run FreeBSD for quite some time.

Remove EISA bus support for add-in cards (https://svnweb.freebsd.org/changeset/base/313839).

The Extended Industry Standard Architecture was a bus standard for IBM PC-compatible computers that was announced in 1988 as an alternative to IBM's proprietary Micro Channel Architecture. Despite being backwards compatible with ISA and not a proprietary bus, EISA never really became popular on desktop computers and was only reasonably successful in the server market, as it was well-suited for bandwidth-intensive tasks.

Remove bdes(1) (https://svnweb.freebsd.org/changeset/base/313329).

The bdes utility implements all DES modes of operation described in FIPS PUB 81, including alternative cipher feedback mode and both authentication modes. The use of DES for anything is discouraged, especially with a static IV of 0, but If you need bdes(1), it can still be installed from the ports system.

Remove the ie(4) driver for Intel 82586 ISA Ethernet adapters (https://svnweb.freebsd.org/changeset/base/304513).

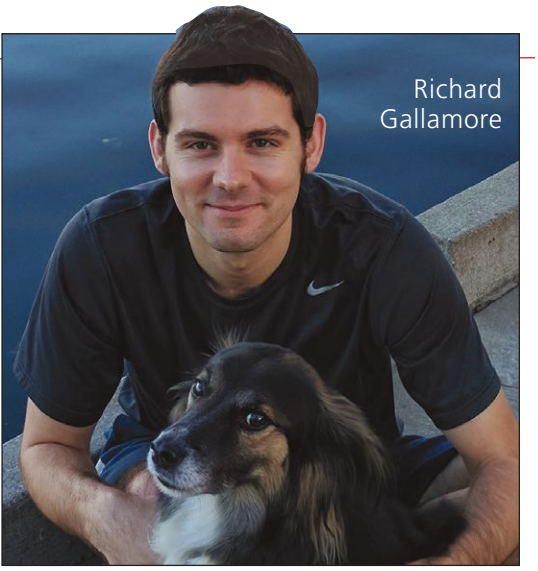
This driver only supports 10-Mb Ethernet using Peripheral Input/Output, and there are not any PC Card adapters supported by this driver, only ISA cards. ●

STEVEN KREUZER is a FreeBSD Developer and Unix Systems Administrator with an interest in retro-computing and air-cooled Volkswagens. He lives in Queens, New York, with his wife, two daughters, and dog.

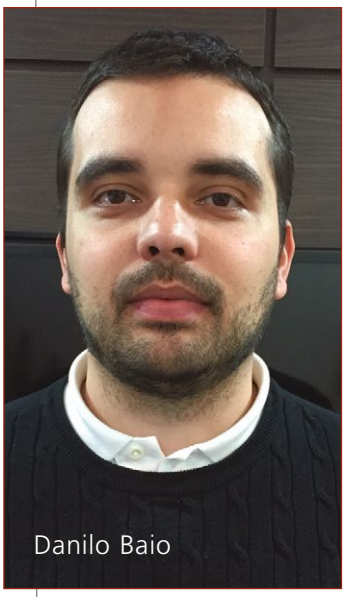
SEE TEXT ONLY

new faces of FreeBSD

BY DRU LAVIGNE


Richard Gallamore

This column shines a spotlight on contributors who recently received their commit bit and introduces them to the FreeBSD community. This month's spotlight is on Danilo Baio and Richard Gallamore, who each received a ports commit bit in May.


Danilo Baio

Tell us a bit about yourself, your background, and your interests.

• Danilo: I am 31 years old and was born in Maringá, Paraná, Brazil. My family is from Marialva and I grew up there. I graduated in Information Systems in 2007 (UniCesumar) and after that I did some specialization in Network Computing (UEL) and IT Project Management (FCV).

I have been interested in computers since I was a kid, and became more comfortable in that area when I started working half-time in a small computer store for my father, and with my brother, who taught me about hardware.

Nowadays, I am not doing many sports, but in my adolescence I swam in state championships. I still really like swimming.

• Richard: My name is Richard Gallamore, and I was born in Kissimmee, Florida. I am 30 years old and have lived in Florida my entire life. I recently moved to San Francisco, California, to start my career in IT. As for my job, I'm currently in transition to my new career. Previously I worked with UPS for 11 years, 4 years as a driver. I also performed side jobs over the years, helping businesses get started and fixing any issues they might have with workstations or printers, and helping with other tasks such as administrating servers, websites, and other various services.

As for my background, well, this field has always been more of a hobby than a necessity. Most of my spare time has been dedicated to computer systems in some form or fashion. Around 2004/2005, I assembled my first computer, and that was quite an interesting experience.

Most of my knowledge came from buying components and learning what they do—creating an environment that I have no need for, but for the sole purpose of understanding how it works. I also often helped friends and family with the basics like networking, printers, and "my computer is running slow," etc. I touched Linux for the first time around 2007, but had few interactions and limited understanding.

Some of my other interests are: keeping systems updated, networking, security, food, Unix (obviously!), working out, or at least staying in shape. Virtual machines are always intriguing, and I absolutely love bhyve! On occasion, I enjoy a video game, though they rarely hold my interest.

How did you first learn about FreeBSD and what about FreeBSD interested you?

• Danilo: After graduating, I started working at BS2 (ISP / Hosting), where I still work, dividing tasks

between BS2 and CRMall (Software Company).

My manager asked if I knew anything about FreeBSD. I said no, and he told me to take a look into it because at BS2 they were running several machines with FreeBSD.

I remember that the first thing that caught my attention was the organization: for instance, separating configuration files of the base system (/etc) from the packages installed through the ports (/usr/local/etc), and then the simplicity of upgrading your system, even major version upgrades.

I also liked the level of commitment of the developers and the Project in keeping the changes well documented and with no surprises for users. I heard about POLA (Principle of Least Astonishment https://www.freebsd.org/doc/handbook/freebsd-glossary.html#pola-glossary) a little before becoming a committer and it all makes sense.

- Richard: Well, in 2011 I decided I needed a storage machine and started using FreeNAS. At this time, I had no idea what FreeBSD was or even that FreeNAS was built on top of FreeBSD. After about a year of using FreeNAS, I got frustrated, because I wanted to use features FreeNAS didn't support or which needed a kernel rebuild, but I couldn't because I was using an older version of FreeBSD and rebuilding the kernel proved challenging on FreeNAS. I finally decided to transition to FreeBSD, and I never looked back!

I also want to note that FreeNAS is a good product and I'm not insulting the distro; it just did not fit my learning personality. FreeBSD in this respect is the perfect operating system for me. It is bare bones and the default install is configured to do almost nothing except the bare minimum to operate. So how does FreeBSD interest me? It suits my personality and knowledge which is a big win for me!

How did you end up becoming a committer?

- Danilo: In 2009, I watched a presentation by Renato Botelho (garga@) about VuXML (https://vuxml.freebsd.org/) and realized that a port we used at BS2 had a security issue. That's when I sent my first PR (https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=141318).

After a few years, I started sending patches to update ports with frequency. After STAGEDIR support was introduced, ports that did not have STAGEDIR support would be removed.

I also sent some contributions to the pfSense project, and in the middle of working between FreeBSD and pfSense, Renato asked me if he could propose my name for a commit bit. If I was accepted, he would be my mentor. This happened in May of this year and I am still learning more day by day.

- Richard: One of my interests has always been keeping systems and software up-to-date, obsessively and sometimes unnecessarily, probably more of an OCD problem. Around 2013/2014 I started learning more about ports, manually updating ports, and just building them. I came across a port called Seafile that was broken. I spent many hours trying to compile it and finally made it usable and wanted to update the port, but had no idea how any of that worked. Luckily, I found some help on iRC and was pushed in the right direction.

During this time, I built up my knowledge on FreeBSD. I was asked a couple of times by FreeBSD developers over this period if I was interested in becoming a committer, but I didn't really have the time due to working way too much and I just didn't think I was ready or had the knowledge for it.

Earlier this year, I finally concluded that I wasn't content with my current lifestyle and needed change. So, I decided to make a career change and moved to California. I messaged the committer who often committed my work, and here I am!

How has your experience been since joining the FreeBSD Project? Do you have any advice for readers who may be interested in also becoming a FreeBSD committer?

- Danilo: I was welcomed by developers from all over the world. This was really cool, and most of them told me to have fun and that's what I'm doing.

I am now more amazed by the level of organization; any small detail matters, even if it is a simple keyword in bugzilla.

In each PR that I work on, I verify whether the contributor is registered in the Additional FreeBSD Contributors list (https://www.freebsd.org/doc/en/articles/contributors/contrib-additional.html). If not, I include them there to show that any contribution is important/relevant, and maybe in the future they could be a FreeBSD committer as well.

My advice for those who might be interested in

becoming a committer is to always think of the other users before sending any patches. Think about whether it makes sense to most of the users or just to you.

Subscribe to the mailing lists related to your areas of interest and keep an eye on the FreeBSD Phabricator (https://reviews.freebsd.org/). There are a lot things going on there, and sometimes you learn something that is not yet in the Handbooks. Usually the mentoring work of new developers goes through there as well.

Also, contributors can submit revisions to Phabricator and then register a PR in bugzilla (http://bugs.freebsd.org/) to have their code committed. There's always a developer willing to take a look and make some comments.

Finally, keep sending patches, be patient, and do not get discouraged if it takes a while to get some feedback. If this happens, it does not mean that you did something wrong, just that the developers are busy or focused on other things, as usually everyone works in their spare time.

I participated in a PR (171246 for anyone who is interested) where this happened. It took seven months to be committed (by me) after becoming a committer.

- Richard: My experience since joining the FreeBSD Project has been great! There are many developers that are extremely knowledgeable.

One of the most important things on the path to becoming a committer is contributions. Not only actively contributing to the Project, but running QA and filling out bug reports completely. Writing a summary of why this patch is needed with the changelog linked or attached. Providing the report with the QA. Doing this shows that one cares about the work and is willing to go the extra mile. Over time it will be noticed and appreciated.

Being active on the mailing list or IRC is also a good way to learn more and help the community. •

DRU LAVIGNE is a doc committer for the FreeBSD Project and Chair of the BSD Certification Group.

SEPTEMBER 2017 BY DRU LAVIGNE

9/17 Events Calendar

The following BSD-related conferences will take place during September 2017.

vBSDCon • Sept 7–9 • Reston, VA

http://vbsdcon.com/ • **https://wiki.freebsd.org/DevSummit/20170907** • The third biennial vBSDCon event will take place at the Hyatt Regency in Reston. This conference brings together members of the BSD community for a series of roundtable discussions, educational sessions, best practice conversations, and networking opportunities. There will be a FreeBSD DevSummit the day before the conference. There is a modest registration fee to attend the conference.

EuroBSDCon • Sept 21–24 • Paris, France

https://2017.eurobsdcon.org/ • **https://wiki.freebsd.org/DevSummit/201709** • The annual European BSD conference will be held at the Espace Saint Martin in Paris. There will be a FreeBSD DevSummit the day before the conference, tutorials and presentations during the conference, and an opportunity to take the BSDA certification exam.

Ohio LinuxFest • Sept 29 & 30 • Columbus, OH

www.ohiolinux.org • The 15th annual Ohio LinuxFest will take place at the Hyatt Regency in Columbus. This open-source event will include several BSD-related talks and a FreeBSD booth in the expo area. This event is free to attend.